Titre: Title:	Exploration de techniques de modélisation et de vérification logicielle en avionique
Auteur: Author:	Jean-François Thibeault
Date:	2006
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Thibeault, J.-F. (2006). Exploration de techniques de modélisation et de vérification logicielle en avionique [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7835/

UNIVERSITÉ DE MONTRÉAL

EXPLORATION DE TECHNIQUES DE MODÉLISATION ET DE VÉRIFICATION LOGICIELLE EN AVIONIQUE

JEAN-FRANÇOIS THIBEAULT
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2006

Your file Votre référence
ISBN: 978-0-494-25579-7
Our file Notre référence
ISBN: 978-0-494-25579-7

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

EXPLORATION DE TECHNIQUES DE MODÉLISATION ET DE VÉRIFICATION LOGICIELLE EN AVIONIQUE

présenté par: THIBEAULT Jean-François

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

Mme. NICOLESCU Gabriela, Doct., présidente

M. BOIS Guy, Ph.D., membre et directeur de recherche

Mme. BOUCHENEB Hanifa, Doctorat, membre

REMERCIEMENTS

Tout d'abord, j'aimerais remercier mon directeur de recherche, Professeur Guy Bois, pour m'avoir mis en contact avec les bons interlocuteurs pour réaliser mon projet de recherche. De plus, j'ai apprécié beaucoup sa confiance qu'il a eu en moi, en me permettant d'expérimenter les charges de laboratoires et les charges de cours.

Un merci tout particulier à Maxence Vandevivère. Sans ses conseils techniques, ses séances de formation et ses commentaires, ma compréhension de SCADE Suite aurait été beaucoup diminuée et la qualité de mon projet affectée. De plus, il a toujours pu organiser des arrangements avec Esterel Technologies pour chacune de mes nombreuses demandes.

J'aimerais souligner l'apport de Pierre Labrèche, qui a lancé mon projet de maîtrise et qui m'a permis d'obtenir le contact industriel nécessaire à sa réalisation, et de Tarek Sabanekh, pour son aide sur le projet CARP.

Je ne pourrai jamais oublier la merveilleuse complicité de mes camarades de travail du CIRCUS (en ordre chronologique): Simon, Morti, François, Pat et Francis. Sans négliger mes confrères de SPACE Codesign (en ordre alphabétique): Ahmed, Cédric, Jérôme, Laurent, Luc, Maxime et les techniciens du GRM: Alexandre et Réjean.

Je suis très reconnaissant envers mes parents, Richard et Sylvie, et ma soeur, Véronique, qui m'ont toujours apporté leur soutien maximal durant mes études et pour chacun de mes projets.

Pour finir, un gros merci à ma copine Tanja pour m'avoir encouragé, pour avoir cru en moi, mais surtout, pour s'être occupée de ma santé intellectuelle et physique durant ces derniers mois. Je l'apprécie beaucoup!

RÉSUMÉ

Ces dernières années, l'accroissement de l'utilisation des systèmes logiciels dans les aéronefs, l'augmentation de la complexité de ceux-ci et l'application de normes strictes définissant le processus de conception de ce type de logiciel a fait augmenter dramatiquement les coûts de conception. C'est la raison pour laquelle des outils permettant de réaliser, d'une manière plus efficace, plusieurs étapes du flux de conception sont apparus sur le marché. L'outil SCADE Suite de la firme Esterel-Technologies en est le parfait exemple. Pour ce faire, SCADE offre un langage de programmation formel garantissant le déterminisme, un simulateur, un générateur de code source qualifiable et des outils avancés de vérification comme un analyseur de couverture structurelle et un module permettant d'effectuer des preuves formelles.

L'utilisation de fonctionnalités avancées de vérification (preuves formelles, couverture structurelle) dans un outil permettant de couvrir une très grande partie du flux de conception d'applications critiques, comme l'outil SCADE Suite, est une possibilité relativement nouvelle sur le marché. Le projet présenté dans ce mémoire consiste à proposer une méthodologie de vérification qui respecte les objectifs de vérification de la norme DO-178B, définissant le processus de conception logicielle en avionique. Cette méthodologie permet l'utilisation des possibilités avancées de vérification offertes par SCADE Suite.

La méthodologie de vérification est basée sur une série d'activité permettant d'atteindre des objectifs de vérification. Ceux-ci ont été déterminés après une analyse des possibilités offertes par SCADE, de sa méthodologie de conception et du chapitre 6 de la DO-178B sur la vérification.

La méthodologie a été éprouvée à l'aide d'une fonctionnalité d'un exemple industriel et comparée à une méthodologie plus traditionnelle, n'utilisant que des tests pour

faire la vérification. Ceci a permis de démontrer les avantages tirés de l'utilisation des outils avancés de vérification. Ces avantages comportent la diminution du nombre de tests, l'augmentation de la couverture des cas possibles d'exécution et la preuve mathématique que certaines situations d'exécution ne peuvent pas se produire.

De plus, une expérience d'utilisation a été réalisée. Celle-ci permet de démontrer qu'aucune perte de productivité n'est associée à l'utilisation de la méthodologie de conception avec l'outil SCADE comparativement à une méthodologie plus traditionnelle en C. Cette expérience, qui a placé les participants dans un contexte industriel d'affectation à un nouveau projet, n'a pas réussi à démontrer un avantage de productivité pour l'une ou l'autre des méthodologies. Et ce, même si les participants avaient beaucoup plus d'expérience de programmation en C qu'avec l'outil SCADE.

Ce projet constitue une excellente introduction à la conception des logiciels critiques en avionique et aux possibilités offertes par l'outil SCADE Suite.

ABSTRACT

In recent years, the increasing use of software systems in aircrafts, the growth of their complexity and the application of strict norms defining the software development process have led to higher development costs. For this reason, new tools specializing in many parts of the software development process have come to the market. SCADE Suite from Esterel-Technologies is the perfect example of such a tool. Indeed, SCADE offers a formal programming language that can guarantee determinism, a simulator for testing, a qualified source code generator and some advanced verification possibilities such as a model test coverage tool and a formal proof engine.

The use of advanced verification techniques (formal proof, structural coverage) in a tool that addresses an important part of the critical software development process, like SCADE Suite, is a relatively new in the industry. In this project, a verification methodology, which respects the DO-178B's verification objectives, avionic's software development standard, will be proposed. The proposed methodology will allow the use of SCADE Suite's advanced verification tools.

The verification methodology is based on activities which address DO-178B's verification objectives. Those have been determined after an analysis of SCADE's possibilities, SCADE's proposed development methodology and after the chapter 6 of the DO-178B which discuss the software verification process.

The methodology has been tried on a software function from a real avionic project and compared to a more traditional methodology based solely on tests. This has shown the advantages in using the advanced verification tools. Those are: the diminution of unit tests, the detection of corner case bugs and the mathematical proof that some situations will never happen.

Furthermore, a usability experiment has showed that no productivity loss is associated

with the use of SCADE's development methodology in comparison with a more traditional C methodology. This experiment, which tried to simulate an assignment to a new development project to the participants, was not able to show a productivity gain between any of the methodologies, notwithstanding that none had previous experience with SCADE and most had a lot of experience with the C language.

This project is an excellent introduction to the critical development process in avionic and to SCADE's possibilities.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES ACRONYMES

ADORA	Analysis and Description Of Requirements and Architecture
ANSI	American National Standards Institute
CARP	Computed Air Release Point
CNRS	Centre National de la Recherche Scientifique
CRP	Point de cheminement CARP
CSML	Compositional State Machine Language
DC	Couverture des Décisions
DMA	Direct Memory Access
DO-178B	Considération sur le logiciel en vue de la certification des systèmes et équipements de bord
DOORS	Dynamic Object-Oriented Requirements System
DSP	Digital Signal Processing
E/S	Entrée/Sortie
ESC	Point de sortie (« Escape Point »)
FAA	Federal Aviation Administration
Fby	Suivi par (« Followed by »)
FORTRAN	Formula Translation/Translator
FPGA	Field Programmable Gate Array
HARP	High Altitude Release Point
ID	Identifiant
IP	Point d'identification
JAA	Joint Aviation Authorities
KCG	Générateur de code qualifiable
MC/DC	Couverture modifiée de condition/décision
MTC	Model Test Coverage

OSEK	Open Systems and the Corresponding Interfaces for Automotive Electronics
PAL	Programmable Array Logic
PBS	Production-Based Specification
PLA	Programmable Logic Array
Pre	Précédant (« Previous »)
ROM	Read Only Memory
RSML	Requirements State Machine Language
RT	Real-Time
RTCA	Radio Technical Commission for Aeronautics
RTL	Register Transfert Level
SAT	Solutionneur de validabilité
SBN	Spécifications de Bas-Niveau
SCADE	Safety Critical Application Development Environnement
SD	Point de ralentissement (« SlowDown Point »)
SHN	Spécifications de Haut-Niveau
SL	Synchronous Language
SML	State Machine Language
SMT	Sundance Multiprocessor Technology Ltd.
SNECMA	Société Nationale d'Etude et de Construction de Moteurs d'Aviation
SPIN	Séparation Incinération
SQRT	Racine carrée
SS	Spécification SCADE
SSM	Safe State Machine
SST	Spécification Système Textuelle

TE	Point de bord de fuite
TNI	Techniques Nouvelles pour l'Informatique
TP	Point de virage (« Turning Point »)
UML	Unified Modeling Language
μC/OS-II	Système d'exploitation temps réel μC, version 2
XTE	Point de cheminement Bord de fuite étendu (« Extended Trailing Edge »)

LISTE DES TABLEAUX

LISTE DES ANNEXES

INTRODUCTION

Les années 70 et les années 80 ont vu apparaître l'intégration de systèmes électriques et numériques dans des projets de pointe comme les centrales nucléaires, les trains et les aéronefs. Ces projets avaient tous comme point commun d'avoir des fonctionnalités critiques qui peuvent mettre en danger la sécurité physique de gens en cas de défaillance.

L'arrivée de ces systèmes électriques et numériques (« fly-by-wire ») dans l'avionique militaire au début des années 70, dans la navette spatiale et dans l'avionique civile (Airbus A320) au début des années 80 a révolutionné le design et le fonctionnement des aéronefs. En effet, ces systèmes ont permis de perfectionner les applications de contrôle à bord des engins volants. Une des conséquences directes de ce perfectionnement fut l'amélioration de la stabilité, de la manoeuvrabilité, de la précision et des performances des aéronefs tout en réduisant la charge de travail des pilotes. De plus, le remplacement des anciens systèmes mécaniques par des systèmes électriques et numériques a permis indirectement, par l'allègement du poids des aéronefs, d'autres améliorations comme la consommation de carburant.

À la fin des années 80 et au cours des années 90, ces systèmes sont devenus incontournables dans la conception de tous projets en avionique, dans le nucléaire et dans le transport. Par contre, la grande diversité de nouvelles fonctionnalités qu'ils ont implémentées a contribué à augmenter la complexité de ceux-ci. C'est la raison pour laquelle il est apparu, durant ces années, des normes définissant le processus de conception, de vérification et du cycle de vie de ces systèmes. Par exemple en avionique civile, la norme DO-178B [46] doit être appliquée lors de la conception de tous projets logiciels.

Problématique

L'augmentation de la complexité des systèmes logiciels critiques et l'apparition de normes strictes définissant le processus de conception de ceux-ci a eu un effet pervers. En effet, les coûts de conception, de maintenance et de certification (la certification est l'attestation officielle de l'application de la norme) ont explosé. C'est pour cette raison que de nouvelles techniques de modélisation plus efficaces ont été développées. Ces techniques, comme les langages de spécification ou la génération de code, permettent d'accélérer, sans perte de qualité, certaines étapes du flux de conception.

Ces dernières années, des outils permettant de réaliser plusieurs étapes du flux de conception de ces systèmes sont même apparus sur le marché. Un de ceux-ci, Esterel SCADE Suite propose plusieurs caractéristiques intéressantes: une méthodologie de conception de systèmes réactifs, systèmes qui réagissent à leur environnement, à partir d'un langage graphique formel d'inspiration synchrone; un générateur de code qualifiable permettant de produire du code source à la sauce DO-178B; un simulateur pour faire des tests; un vérificateur formel permettant de faire des preuves formelles; un outil pour mesurer la couverture structurelle; des passerelles vers des outils de gestion de spécifications. Le principal défi de ces outils est d'être à la fois intuitif et puissant, en plus de proposer des méthodologies permettant de respecter des normes comme la DO-178B.

L'utilisation de fonctionnalités avancées de vérification (preuves formelles, couverture structurelle) dans un outil permettant de couvrir une très grande partie du flux de conception d'applications critiques, comme l'outil SCADE Suite, est une possibilité relativement nouvelle sur le marché. Dans la littérature, il n'existe pas encore de méthodologie de vérification faisant usage de ces outils avancés, respectant les objectifs de la DO-178B et s'intégrant facilement à la méthodologie de conception de SCADE (qui a été maintes fois éprouvées dans des projets en avionique ces dernières

années). Une telle méthodologie de vérification jumelée à un exemple d'application industrielle de celle-ci pourrait faciliter et maximiser le potentiel d'utilisation de ces nouvelles fonctionnalités avancées de vérification.

Objectifs

L'objectif général de ce projet est l'exploration des possibilités offertes par SCADE. Plus spécifiquement, une méthodologie de vérification qui fait usage des nouvelles fonctionnalités avancées de l'outil de modélisation SCADE Suite sera proposée. Cette méthodologie devra pouvoir s'intégrer à une méthodologie de conception respectant les objectifs de la DO-178B.

Un des objectifs spécifiques est d'essayer de démontrer le caractère intuitif de l'outil SCADE Suite. Le caractère intuitif de SCADE, entre autres grâce à son langage de modélisation graphique, est souvent mentionné comme avantage indirect lors de l'application d'une méthodologie de conception. Il est aussi très important en conception et en vérification, car il permet de réduire les problèmes d'interprétation du modèle.

Méthodologie

Pour répondre au premier objectif spécfique du projet, les possibilités de l'outil SCADE Suite seront tout d'abord explorées. Par la suite, la méthodologie de conception utilisée en industrie dans un contexte de DO-178B sera analysée et comparée à celle proposée par SCADE Suite. Ensuite, la section de la DO-178B [46], concernant la vérification[1], sera examiné. Cette investigation permettra d'en faire ressortir les objectifs de vérification devant être respectés dans une méthodologie de

[1]Cette section correspond au chapitre 6 de la DO-178B

vérification. Ces objectifs seront alors mis en contexte avec l'outil SCADE Suite. Ceci permettra de proposer une série d'activités répondant à ces objectifs. Les activités seront alors intégrées à une méthodologie de vérification. Par la suite, la méthodologie sera mise en pratique à l'aide d'un exemple d'application tiré d'un projet industriel en avionique. Actuellement dans l'industrie, les tests sont abondamment utilisés lors de la vérification. Donc, pour obtenir une base de comparaison, la méthodologie sera répliquée sur le même exemple d'application, mais cette fois-ci sans les fonctionnalités avancées de vérification de SCADE et donc basée presque entièrement sur les tests.

Pour le second objectif spécifique, une expérience d'utilisation de l'outil SCADE sera menée avec une trentaine de participants. Cette expérience permettra de comparer la productivité des sujets avec l'outil de conception SCADE et avec une approche de conception plus traditionnel en C. Plus particulièrement, celle-ci essaiera d'identifier le type d'approche qui permet aux participants de devenir productifs plus rapidement. Ceux-ci seront mis dans la situation où ils auraient à travailler sur un nouveau projet en industrie sans expérience préalable sur le sujet de celui-ci.

Originalité et contribution

La principale contribution de ce projet est de proposer une méthodologie de vérification utilisant les fonctionnalités avancées de vérification de l'outil SCADE Suite dans un contexte de DO-178B. Avec un exemple d'application industrielle à l'appui, son efficacité et ses limites seront démontrées. En effet, il sera constaté que dans certains types de projets, plus de la moitié des tests peuvent être éliminés et remplacés par des preuves formelles. De plus, ce projet se veut aussi une activité de synthèse des types de méthodologies d'inspiration DO-178B utilisées dans les projets en avionique. Le mémoire peut être utilisé comme une excellente introduction à cet univers relativement spécialisé.

Également, l'expérience d'utilisation est la première, dans la littérature, à comparer la méthodologie SCADE Suite à une méthodologie plus traditionnelle en C pour ainsi vérifier le caractère intuitif de l'outil.

Distribution des chapitres

Ce mémoire est réparti en cinq chapitres. Le premier chapitre présente des concepts, les principaux langages synchrones et de spécification et les outils de modélisation utilisés dans le monde des systèmes réactifs. De plus, une introduction à la certification en avionique et à la norme DO-178B [46] sera effectuée. Le deuxième chapitre fait un survol de l'outil Esterel SCADE Suite, utilisé pour faire de la modélisation de systèmes réactifs critiques en avionique. Le troisième chapitre introduit la méthodologie utilisée dans l'industrie et celle proposée par SCADE pour faire le développement d'applications critiques. Ce chapitre présente aussi les résultats d'une expérience d'utilisation qui vérifie un des principaux avantages de l'outil SCADE Suite, c'est-à-dire son caractère intuitif. Une méthodologie de vérification utilisant les forces et les possibilités de l'outil SCADE Suite et qui respecte les objectifs de la DO-178B est proposée au quatrième chapitre. Finalement, le cinquième chapitre décrit les résultats obtenus lors de la mise en oeuvre de la méthodologie avec une application avionique tirée d'un projet industriel.

CHAPITRE 1

REVUE DE LA CONCEPTION DES SYSTÈMES RÉACTIFS

Ce chapitre présente une revue des concepts, langages et des outils utilisés dans la conception des systèmes réactifs. Cette classe de système logiciel est largement utilisée en avionique.

1.1 Concepts

Cette section présente une introduction aux systèmes réactifs et certains concepts reliés à ceux-ci.

1.1.1 Systèmes Réactifs

Le terme « système réactif » a été introduit par David Harel et Amir Pnueli [33] au milieu des années 80. Ils ont donné aux systèmes réactifs l'image de la boîte noire qui réagit avec son environnement à la vitesse de celui-ci [35].

Les systèmes réactifs interagissent continuellement avec un environnement physique qui est incapable de se synchroniser avec eux (souvent l'environnement ne peut pas attendre). Leur temps de réponse doit respecter les spécifications induites par l'environnement.

Cette classe a été introduite pour se différencier des « systèmes transformationnels », systèmes qui se terminent en produisant un résultat à partir de données initiales (ex. : compilateurs), et des systèmes interactifs, systèmes qui interagissent continuellement

avec un environnement qui possède des mécanismes de synchronisation (ex. : systèmes d'exploitation) [30].

Les systèmes réactifs sont généralement utilisés dans les domaines du contrôle, des processus automatiques, en traitement de signal, dans les protocoles de communication, ou même les interfaces usager qui demandent des temps de réponse courts. Ces systèmes possèdent les caractéristiques suivantes :

- Contrairement aux systèmes interactifs, ils sont souvent utilisés dans des environnements critiques (la vie et la sécurité des gens en dépend) où ils doivent être déterministes.
- Ils s'exécutent en parallèle avec leur environnement.
- Souvent, ils sont distribués pour des raisons de vitesse d'exécution, de tolérance aux fautes, et de fonctionnalité.
- Ils doivent respecter des contraintes temporelles.

Les systèmes réactifs peuvent être vus comme une séquence infinie de vecteurs d'entrée/sortie qui, à chaque pas d'exécution, déterminent des valeurs de sortie à partir de valeurs d'entrée présentes ou passées. Ce point de vue est celui adopté par les langages de programmation synchrones [28].

1.1.2 Déterminisme

Le déterminisme est une caractéristique importante des systèmes réactifs et en particulier ceux qui sont critiques. Un système réactif déterministe produira une séquence de valeurs de sortie identique si sa séquence de valeurs d'entrée est identique. Donc pour un même vecteur d'entrée, le vecteur de sortie sera le même. Ceci est dû au fait que les sorties d'un tel système sont une fonction mathématique des entrées.

Les systèmes déterministes sont généralement plus simples à spécifier, à déverminer et à analyser.

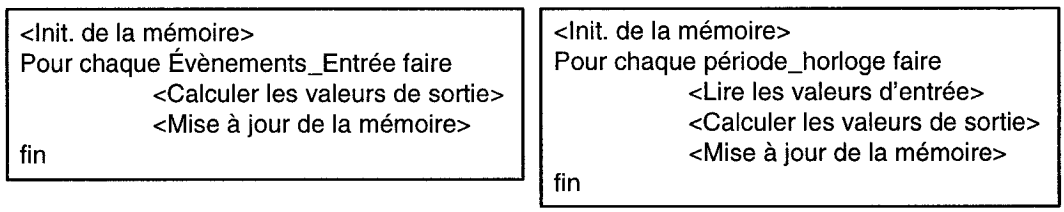
Les systèmes purement séquentiels sont déterministes. Par contre, il ne faut pas seulement associer le déterminisme aux systèmes séquentiels. En effet, il est possible de décomposer la plupart des systèmes réactifs en sous-systèmes concurrents qui exécutent et coopèrent de façon déterministe. Les méthodes synchrones proposent une méthodologie permettant de produire ce type de systèmes [12].

1.1.3 Approche synchrone

L'approche synchrone est basée sur un cadre mathématique qui combine le synchronisme (le temps avance par pas d'exécutions avec une ou plusieurs horloges) et la concurrence déterministe.

En effet, il existe deux modèles d'exécution synchrone de systèmes réactifs. Le premier (voir Figure 1.1a) est le modèle par évènements où une boucle simple réagit à des évènements discrets en entrée pour calculer des vecteurs de sortie. Le deuxième modèle, plus simple, est le modèle échantillonnage (voir Figure 1.1b) où une boucle simple est exécutée à chaque période d'horloge. Dans celle-ci, un vecteur d'entrée est lu et un vecteur de sortie est calculé. La nature des systèmes réactifs suppose que l'exécution de ces boucles simples peut s'effectuer en parallèle. Ceci implique donc de la concurrence.

Ces deux modèles sont habituellement implantés en machines à états finis (ou automates), qui sont très bien connues sur le plan mathématique. Les états sont définis par les valeurs calculées et les transitions correspondent à des réactions. Les réactions impliquent généralement plusieurs calculs considérés atomiques par l'automate. Cette atomicité est une des bases du paradigme synchrone. Une réaction



a) « Par Évènements » b) « Échantillonnage »

Figure 1.1 Modèles d'exécution des systèmes réactifs

atomique est un instant de temps discret (logique ou abstrait) de durée physique nulle [44]. Ainsi, tous les évènements se déroulant durant une réaction sont considérés comme étant simultanés [28], [4].

L'hypothèse de l'atomicité permet de simplifier le raisonnement sur les systèmes réactifs où les réactions ne peuvent s'entremêler (une réaction atomique ne peut pas en interférer une autre). Ainsi, la concurrence entre les réactions n'est plus considérée. Sans cette simplification, la concurrence serait source d'indéterminisme [12].

Les machines à états finis sont des outils très utiles, en raison de leur simplicité, leur puissance expressive et leur efficacité. Par contre, il est très difficile de les programmer « à la main ». C'est pour cette raison que les langages synchrones ont été développés. En effet, ceux-ci offrent des constructions déterministes et une sémantique modulaire de haut niveau facilitant le design de ces machines à états finis [28]. La section 1.2 présentera les principaux langages synchrones.

1.1.4 Flux de données

Dans les domaines du contrôle et de l'électronique, les gens sont habitués à modéliser des systèmes réactifs avec l'aide de « réseaux d'opérateurs » qui transforment des données. Ces systèmes sont modélisés à l'aide d'équations booléennes, des fonctions de transfert avec des structures diagrammes blocs et des équations dynamiques.

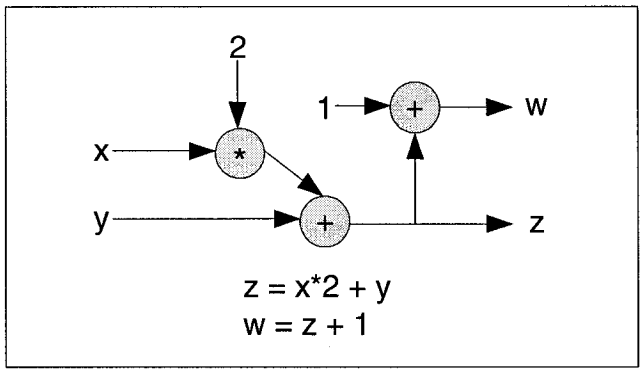


Figure 1.2 Exemple d'une description « Flux de données » et ses équations

Dans le domaine de l'informatique, ce formalisme ressemble beaucoup à celui du « flux de données » (voir Figure 1.2). Lorsqu'utilisé comme base d'un langage haut niveau (langages permettant une expression naturelle d'un problème à l'aide d'un programme), le paradigme « flux de données » possède les avantages suivants [30] :

- C'est un modèle fonctionnel comportant une clarté mathématique sans effets de bord complexes (c.-à-dire la perte du sens mathématique d'une fonction, l'utilisation de variables globales, etc.). En effet, le comportement d'une fonction mathématique est indépendant des entrées. Par exemple, le comportement de la multiplication ne sera pas altéré si les entrées sont permutées. De plus, la multiplication se fait toujours à partir d'entrées du même type (sinon ce n'est plus une multiplication au sens mathématique). Ceci le rend très bien adaptée à la vérification formelle et à la programmation sécuritaire étant donné que le modèle peut être vu comme étant invariant au temps. De plus, il facilite la réutilisation.

- C'est un modèle parallèle, car l'exécution séquentielle n'est causée que par la dépendance entre les données. Ceci permet la dérivation d'implémentations parallèles de données indépendantes.

1.1.5 Flux de commande

Le flux de commande est utilisé pour spécifier des systèmes où les évènements discrets sont très présents. Ces évènements peuvent être produits par des capteurs discrets, des entrées d'usagers ou des évènements internes au programme. Généralement, le contrôle discret est exprimé à l'aide de machines à états ou de langages impératifs (ces langages décrivent les programmes en termes de calculs à effectuer en fonction de l'état de ceux-ci et d'instructions permettant de changer leur état) [35].

1.1.6 Idéologie « Orienté-modèle »

L'idéologie « *Orienté-modèle* » (Model-driven) pour la spécification des systèmes diffère de l'idéologie traditionnelle « *Orienté-document* ». En effet en « *Orienté-document* », les spécifications du système sont faites en texte. Un développeur prend alors ces spécifications et implémente le système. Cette méthodologie peut mener à des problèmes où le développeur peut ne pas implémenter correctement ce qui est présent dans la spécification. De plus, il devient très difficile de valider un modèle avant son implémentation (étant donné que les spécifications ne sont pas exécutables). En « *Orienté-modèle* », on utilise un environnement graphique pour spécifier le système. Ceci permet d'exécuter cette spécification et ainsi vérifier (avant l'implémentation) si elle est correcte. De plus, il est souvent possible de générer du code directement à partir de la spécification. Ceci permet alors d'obtenir un comportement du code qui reste fidèle à la spécification.

Par contre, la plus grande problématique de cette idéologie est d'avoir des moyens pour garantir la relation entre le code et le modèle (par exemple à l'aide d'outils de génération de code qualifiables, voir section 2.4.2.1). Si cette relation n'est pas bien définie, l'application de l'idéologie « *Orienté-modèle* » devient beaucoup moins

intéressante.

1.2 Les langages synchrones

Cette section présentera les principaux langages synchrones qui sont utilisés dans le monde universitaire et dans l'industrie.

1.2.1 LUSTRE

1.2.1.1 Historique

Le développement du langage LUSTRE a commencé en 1984 au CNRS à Grenoble en France par une équipe dirigée par Paul Caspi et Nicholas Halbwachs. Leur but était de proposer un langage simple, basé sur le modèle de flux de données utilisé par la plupart des ingénieurs en contrôle [4]. Depuis ce temps, son développement n'a jamais vraiment cessé. Il a été utilisé dans plusieurs projets industriels (voir section 1.2.1.3) et est devenu le langage de base de l'outil de développement SCADE (voir chapitre 2).

1.2.1.2 Caractéristiques et sémantique

Le langage LUSTRE est un langage déclaratif. C'est-à-dire un langage qui décrit une entité plutôt que de décrire comment la créer, comme dans un langage impératif. Dans le monde des langages synchrone, LUSTRE tombe dans la catégorie de ceux spécialisés dans le flux de données. En effet dans LUSTRE, chaque variable ou expression se défini comme un flux, c'est-à-dire une paire faite à partir de :

- une séquence infinie de valeurs d'un type donné (x_0, x_1, ..., x_n, ...);

- une *horloge* représentant une séquence temporelle.

On défini x_n comme étant la valeur x au n^{ime} cycle d'exécution d'un flux, donc chaque variable prend une valeur à chaque cycle du programme (voir Figure 1.3). Un programme LUSTRE ou une partie de celui-ci, possède un comportement cyclique défini par une horloge qu'on appelle *l'horloge de base.* Cette horloge n'a pas nécessairement de signification physique et est généralement considérée comme étant logique.

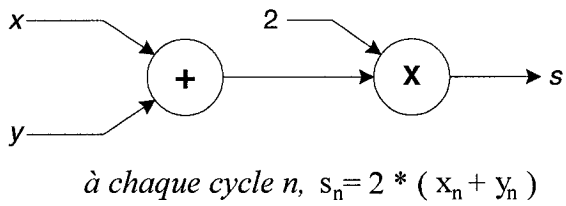


Figure 1.3 Exemple d'une équation flux de données cycliques

Un programme LUSTRE calcule un flux de sortie à partir d'un flux d'entrée. Le flux de sortie est défini à partir d'équations (au sens mathématique) et est considéré comme invariant au temps (c'est-à-dire, l'équation « x=e » correspond à « $\forall n, x_n = e_n$ »). Ceci permet d'exprimer un des principes les plus importants du langage, *le principe de substitution*: X peut être substitué (remplacé) par E partout dans le programme et réciproquement. Une des conséquences de se principe est que les équations peuvent être écrites dans n'importe lequel ordre sans changer la fonctionnalité du programme.

Les opérateurs LUSTRE agissent globalement sur des flux. Par exemple, l'équation « x+y » correspond au flux (x_0+y_0, x_1+y_1, ..., x_n+y_n, ...). En plus des opérateurs arithmétiques, LUSTRE offre une variété d'opérateurs booléens, conditionnels et temporels. Les deux principaux opérateurs temporels agissants sur des flux sont:

- l'opérateur « **pre** » (« *previous* ») donne l'accès à la valeur précédente de son argument: par exemple, « **pre(x)** » correspond au flux (*nil*, x_0, ..., x_{n-1}, ...), où la première valeur *nil* n'est pas définie (« *non initialized* »).

- l'opérateur « -> » (« *suivi de* ») est utilisé pour définir des valeurs initiales: par-exemple « **x -> y** » correspond au flux (x_0, y_1, ..., y_n, ...), qui initialement est égal à x, et à y par la suite.

Un système d'équations défini sous LUSTRE peut être encapsulé dans une sous-routine réutilisable. Ces sous-routines sont appelées « noeuds ». Un noeud possède une interface composée d'un vecteur d'état. Le vecteur d'état est constitué d'un vecteur d'entrée (flux d'entrée), d'un vecteur de sortie (flux de sortie) et de mémoires (pour garder les anciennes valeurs, les valeurs locales, etc.). Les valeurs prises par le vecteur de sortie ne sont déterminées que par le vecteur d'entrée, les valeurs passées de celui-ci et par des variables locales. Ceci garantit un déterminisme dans l'exécution du noeud. Les noeuds permettent à LUSTRE de décrire des programmes de façon hiérarchique et de faciliter la réutilisation.

Chaque noeud d'un programme LUSTRE peut s'exécuter à son propre rythme, c'est-à-dire avoir sa propre horloge. Ces horloges « locales » fonctionnent à un rythme plus lent que l'horloge de base, qui est l'horloge absolue du système. Ces horloges « locales » sont implantées à l'aide d'opérateurs booléens qui permettent l'activation ou la non-activation d'un noeud [4], [29], [28].

Pour s'assurer qu'un modèle LUSTRE aura un comportement fonctionnel déterministe, le langage ne permet pas d'avoir des boucles à délai nul. Cette méthode acyclique permet au modèle de rester synchrone, sans qu'il souffre de problèmes de concurrence.

1.2.1.3 Utilisation industrielle

Le contexte industriel européen et français des années 80 a permis à LUSTRE et aux autres langages synchrones (voir Section 1.2.2 et 1.2.3) de s'illustrer. LUSTRE a été impliqué dans le système SPIN de la nouvelle génération des réacteurs nucléaires

français, les N4. C'était la première fois que les fonctions les plus critiques du système étaient réalisées par un ordinateur. Quelques années plus tard, l'outil SCADE (voir chapitre 2), basé sur LUSTRE, a été développé par la firme Verilog et a été utilisé pour faire une partie du logiciel de contrôle de vol de l'Airbus A340-600 et pour la ré-ingénierie du système de contrôle des rails du métro de Hong Kong [4], [29].

1.2.2 ESTEREL

1.2.2.1 Historique

Historiquement, le langage ESTEREL est considéré comme le premier langage synchrone. Une première notation informelle a été développée en 1984 par une équipe, de l'École des Mines de Paris à Sophia Antipolis, composée de Jean-Paul Marmorat et Jean-Paul Rigault. La sémantique formelle du langage et une première génération d'outils de vérification formelle ont été développées sous la gouverne de Gérard Berry [24].

1.2.2.2 Caractéristiques et sémantique

Le langage ESTEREL est un langage impératif spécialisé dans les descriptions de flux de commande. Suivant une sémantique mathématique, le langage peut être compilé efficacement en machine à états finis. Chacune des réactions d'un programme ESTEREL doit être représenté par une fonction déterministe de la forme :

$$(state,\ input) \mapsto (next\ state,\ output).$$

Ceci implique que chaque réaction aura une solution unique [4]. Cette méthode, dite de solution unique point fixe, est utilisée par le compilateur pour s'assurer que le modèle reste synchrone et qu'aucun problème de concurrence ne puisse survenir.

Comme certains autres langages concurrents (voir OCCAM [56]), ESTEREL possède un opérateur (« || ») représentant le parallélisme. Avec cet opérateur, des entités parallèles peuvent être directement programmées. Un programme ESTEREL est constitué d'entités s'exécutant de manière synchrone à une horloge globale.

ESTEREL possède un seul mécanisme de communication, le « *broadcast* ». En effet, lorsqu'une entité veut communiquer avec d'autres, celle-ci « lève la main » (c'est à dire fait un « broadcast ») et ainsi, toutes les entités peuvent la voir. Ceci est en opposition aux protocoles « handshaking » où les communications sont faites d'une entité à une autre sans que des entités en dehors de la communication puissent voir ce qui se passe. Les communications sont réalisées en utilisant des **signaux** (*signal*) qui peuvent être émis (*emit*), testés si présent (*present*) ou non et auxquels une valeur peut être ajoutée. Les entrées et les sorties des modules sont des signaux.

```
present S then emit T end
||
emit S
||
present S then emit U end
```

Figure 1.4 Exemple d'entités en parallèle

Les « broadcasts » sont limités aux « instants ». En effet, lorsqu'un signal est émis, celui-ci sera seulement vu comme « présent » par toutes les autres entités durant cet instant. Par exemple à la Figure 1.4, le signal **S** est émis et sera vu « présent » par les deux autres entités, qui à leur tour émettront les signaux **T** et **U**. Les signaux **S**, **T** et **U** seront alors émis au même instant et considérés comme *synchrones*. Il faut noter qu'en raison de sa nature déterministe, le langage ne permettra pas qu'un signal soit « présent » et « absent » durant un même instant.

Le mécanisme de « broacasts » instantanés des signaux est une fonctionnalité très puissante du langage, mais ceci peut mener à des problèmes de causalité. Les

problèmes de causalité sont au monde synchrone ce que les « deadlocks » sont au monde asynchrone. En examinant le programme à la Figure 1.5, on peut y voir que le programme a deux solutions. Dans la première, S1 est présent et S2 est absent. Dans la deuxième, S1 est absent et S2 est présent. Donc, ce programme est non déterministe et viole l'hypothèse de déterminisme d'ESTEREL. Il faut savoir que ces problèmes de causalité sont détectés par le compilateur. Ceci est en fait la grande différence avec les « deadlocks » qui ne sont souvent détectés qu'à l'exécution [12].

1.2.2.3 Utilisation industrielle

Plusieurs versions de compilateurs ont été développées au cours des dernières années. Les trois premières versions se sont spécialisées dans la transformation du code ESTEREL en automate. Ces premières versions ont été utilisées, entre autres, par Dassault Aviation pour la réalisation de programmes de contrôle de vol. À partir de la version 4 du compilateur, on a commencé à traduire le code ESTEREL en logique numérique. À partir de ce moment, ESTEREL a été utilisé pour la programmation de FPGA, le développement de puces DSP et pour la vérification matérielle. Tout comme son cousin LUSTRE, ESTEREL est devenu la base d'un outil de développement industriel, Esterel Studio (voir Section 1.5.1) développé à partir de 1998 [4], [9].

```
Signal S1, S2 in
  present S1 else emit S2 end
||
  present S2 else emit S1 end
end
```

Figure 1.5 Exemple d'un problème de causalité

1.2.3 SIGNAL

1.2.3.1 Historique

Le langage SIGNAL a été développé en 1984 par Albert Benveniste et Paul Le Guernic à l'Inria à Rennes en France. SIGNAL a été conçu pour permettre la spécification des systèmes. Contrairement aux programmes, les systèmes doivent être construits de sorte qu'il soit possible de faire des mises à jour de leurs composants, d'en ajouter et d'en supprimer. En bref, les systèmes doivent être considérés comme *ouverts* [4].

1.2.3.2 Caractéristiques et sémantique

Le langage SIGNAL est, comme son cousin LUSTRE, un langage déclaratif spécialisé dans la description du flux de données. SIGNAL met l'accent sur une programmation « par évènements », tandis que LUSTRE se spécialise dans la programmation « par échantillons » [29], [28]. Tout comme LUSTRE et ESTEREL, SIGNAL base toute sa sémantique sur une représentation mathématique déterministe.

L'exécution d'un programme synchrone, P, est souvent simplifiée par la formulation que celui-ci fait quelque chose à chaque réaction. Ainsi, l'horloge d'activation de P correspond à ses exécutions successives. Ceci est le point de vue pris par les modèles d'exécution de la Figure 1.1. En effet dans le modèle « par évènements », au moins un évènement sur les entrées du module est nécessaire pour produire une réaction. Dans le modèle « par échantillons », les réactions sont effectuées à chaque coup d'horloge. Par contre, le programme P peut être utilisé dans un environnement qui doit être activé à des moments ou P ne doit pas l'être. Le langage SIGNAL prend le point de vue que l'horloge locale d'activation de P doit être distincte de la séquence « ambiante » des réactions du système. En effet, les réactions ambiantes doivent être

invisibles à P. Cette approche de SIGNAL en fait un langage « multihorloge » [4].

Dans le langage SIGNAL, chaque signal possède sa propre horloge. L'horloge du programme devient l'horloge suprême des horloges de chaque signal. Chaque signal est une séquence typée de valeurs (booléen, entier, réel). De plus, la valeur $\perp$ est ajoutée aux domaines des signaux pour indiquer la présence ($\neq \perp$) ou l'absence ($= \perp$) de ceux-ci dans une réaction. L'usager ne peut pas manipuler cette valeur, car celui-ci en viendrait à contrôler la séquence « ambiante » des réactions. Si on observe l'équation suivante:

Y := X when B

où Y devient présent quand B est vrai et que X est présent, on obtient le comportement suivant:

X:	1	2	$\perp$	3	4	5
B:	V	F	V	F	V	$\perp$
Y:	1	$\perp$	$\perp$	$\perp$	4	$\perp$

Donc ici, des opérations sont effectuées sur des signaux absents (c.-à-d. d'horloges différentes). Grâce à ce comportement, SIGNAL peut supporter les deux modèles d'exécutions de la Figure 1.1. SIGNAL permet au designer de mixer les communications *réactives* (celles offertes par l'environnement) et les communications *proactives* (celles demandées par le programme). Tout en offrant les mêmes possibilités de calculs sur les données que LUSTRE, SIGNAL est plus général que ce dernier en ce qui concerne les horloges [36].

Dans l'exemple précédent, l'horloge de Y est plus lente que l'horloge de X. Contrairement aux autres langages synchrones, SIGNAL offre un mécanisme pour combiner les horloges de signaux et ainsi créer un signal avec une horloge plus grande que celle des signaux qui le compose. Si on observe l'équation suivante:

W := U default V

où W devient présent quand U est présent ou sinon lorsque V est présent, on obtient le comportement suivant où W a une horloge plus rapide que U et V:

V:	1	$\perp$	2	3	$\perp$	5
U:	$\perp$	5	6	7	4	$\perp$
W:	1	5	2	3	4	5

Deux signaux sont considérés comme possédant la même horloge lorsque leur séquence de présence et d'absence de valeur est identique [5]. Contrairement à SIGNAL, LUSTRE ne permet pas que l'horloge « de base » du système soit séparée en intervalles plus petits [30].

Tout comme LUSTRE, les systèmes SIGNAL sont modélisés à l'aide de diagrammes blocs. Des sous-blocs sont connectés entre eux et hiérarchiquement avec un bloc de haut-niveau. Chaque bloc possède des signaux sur lesquels des opérations mathématiques et temporelles peuvent être effectuées [4].

1.2.3.3 Utilisation industrielle

À ses débuts, SIGNAL a été développé en collaboration avec le Centre National d'Études des Télécomunications et a été utilisé pour modéliser des applications de traitement de signaux s'exécutant sur des DSP (digital signal processors). Plus tard dans les années 90, la compagnie Snecma s'est mise à utiliser SIGNAL pour modéliser des systèmes de contrôle de moteur d'avion. En 1993, l'outil de modélisation Sildex, basé sur SIGNAL, a été développé par TNI (Techniques Nouvelles pour l'Informatique). Cet outil a été largement utilisé par Snecma [4].

1.2.4 CSML

CSML (Compositional State Machine Language) est un langage impératif synchrone basé sur le langage SML. Le langage est aussi basé sur l'hypothèse synchrone où les

actions du programme sont considérées comme étant instantanées. CSML est utilisé pour décrire des contrôleurs matériels basés sur des machines à états finis. En effet, les programmes CSML peuvent être compilés dans des tables de transitions d'états qui peuvent par la suite être implémentés dans des PAL's, des PLA's et des ROM's. Le langage comporte la plupart des constructions des autres langages impératifs, soit des boucles, des conditions et des opérateurs d'exécution en parallèle. Pour plus d'informations, référer à [17].

1.2.5 Autres langages synchrones

SL est un langage synchrone basé sur ESTEREL qui propose une nouvelle hypothèse sur l'absence et la présence de signaux [13]. LAVA est un langage de description pour le matériel qui s'inspire des langages synchrones [42]. Il est spécialisé dans le flux de données. PBS (Production-Based Specification) est un langage applicatif de description matérielle d'inspiration synchrone spécialisé dans le flux de commande [48].

1.2.6 Comparaison des langages

Le Tableau 1.1 donne un résumé des principales caractéristiques de chacun des principaux langages synchrones présentés dans cette section.

1.3 Modèles graphiques

Cette section présentera différents modèles graphiques utilisés pour représenter le flux de commande.

Tableau 1.1 Comparaison des principaux langages synchrones

	LUSTRE	ESTEREL	SIGNAL
Langage déclaratif/impératif	*déclaratif*	*impératif*	*déclaratif*
Spécialisé pour flux commande/données	*données*	*commande*	*données*
Sémantique mathématique	*oui*	*oui*	*oui*
Exécution déterministe	*oui*	*oui*	*oui*
Possibilité d'avoir des horloges locales plus lente et/ou rapide que l'horloge du système	*lente*	*lente*	*lente/ rapide*
Permet d'exprimer le parallélisme à l'aide d'un opérateur	*non*	*oui*	*non*
Mécanisme principal de communication intermodule	*entrées/ sorties*	*broadcasts de signaux*	*signaux*
Méthode pour garantir le déterminisme sans problèmes de concurrence entre modules	*Acyclique*	*Équation point-fixe unique*	*Relations et contraintes*
Possibilité de décrire des fonctions séquentielles	*oui, opérateur « pre »*	*non*	*oui, opérateur « $ »*

1.3.1 STATECHARTS

STATECHARTS est un langage de spécification graphique possédant une sémantique essayant d'obéir à l'hypothèse synchrone [55]. Le principe du langage est d'utiliser des machines à états pour modéliser des systèmes. Les boîtes sont utilisées pour exprimer des états et les flèches pour des transitions. Chacune des transitions peut être marquée par un évènement, E, et une condition, C. Une transition sera prise lorsque l'évènement E se produira si et seulement si la condition C est respectée. Un des avantages du langage est de permettre l'encapsulation d'états dans un état hiérarchique. Ceci permet de réaliser des transitions directement à partir de cet état hiérarchique et d'ainsi alléger le modèle. La Figure 1.6 montre un exemple d'états encapsulés où la transition $e_3(c_3)$ sera prise si le système se trouve dans un des états encapsulés de *SD*, soit *SA* ou *SB* et si l'évènement e_3 respectant la condition c_3

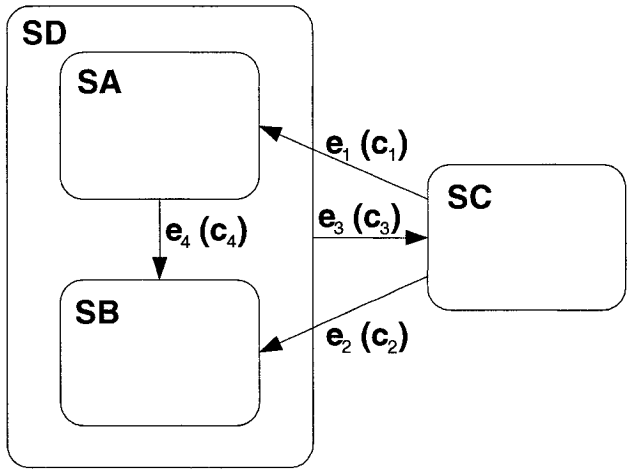


Figure 1.6 Exemple d'une modélisation statecharts

survient. Ici, l'état hiérarchique SD permet de factoriser la transition $e_3(c_3)$, celle-ci devenant alors une propriété commune des états SA et SB.

STATECHARTS offre plusieurs constructions pour améliorer et faciliter la modélisation. Une de celles-ci est l'opérateur « historique ». Cet opérateur permet de garder en mémoire l'état dans lequel on était avant de faire une transition. Ceci devient utile lorsqu'une transition nous fait « quitter » un état hiérarchique et qu'on veuille avoir la possibilité de revenir plus tard au même endroit.

Le langage permet aussi l'exécution d'états en parallèle. À partir d'un état hiérarchique, un système d'états fonctionnant en parallèle peut être créé. La Figure 1.7 montre un exemple de ce cas. Il est à noter que le rond noir avec la flèche indique l'état initial. Lorsqu'une transition mènera à l'activation du super état SK, ceci activera en parallèle les états SI et SJ. Dans SI, l'état initial sera SE, et SG sera l'état initial de l'état hiérarchique SJ. Pour plus d'informations sur les possibilités offertes par STATECHARTS, référer à [32].

Les compilateurs STATECHARTS ne permettent pas des constructions qui pourraient mener à des *inconsistances*. Ces *inconsistances* sont des situations où des signaux

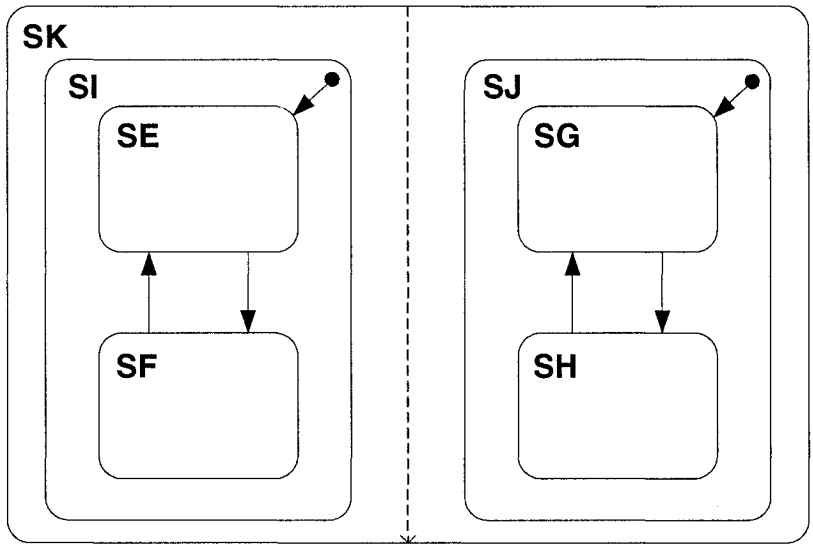


Figure 1.7 Exemple d'une modélisation statecharts parallèle

se retrouvent à la fois absents et présents. Toute situation non déterministe sera détectée par le compilateur. Celui-ci fera une recherche exhaustive pour toutes les situations problématiques et ne permettra que les « bonnes » constructions [55]. Par contre, les STATECHARTS ne respectent pas totalement les hypothèses synchrones et il existe plus d'une vingtaine de sémantiques différentes [39], [1]. La principale force de ce langage (tout comme SYNCCHARTS à la prochaine section) est son formalisme au sens mathématique.

1.3.2 SYNCCHARTS

Les SYNCCHARTS ont été introduits comme une forme graphique du langage ESTEREL. Ceux-ci s'appuient donc sur la même sémantique mathématique. La syntaxe des SYNCCHARTS a été influencée par les STATECHARTS (voir Section 1.3.1). En fait, ce double héritage STATECHARTS/ESTEREL fait en sorte qu'un modèle SYNCCHARTS ressemble à un modèle STATECHARTS mais son comportement est différent. Les SYNCCHARTS permettent, malgré leur apparente simplicité,

l'expression de comportements complexes qui reflètent l'hypothèse synchrone.

Avec les SYNCCHARTS, un état peut être simple (représenté par une ellipse) ou un macroétat (représenté par un rectangle). Ce macroétat ressemble beaucoup à un état hierarchique de STATECHARTS. Par contre, les transitions (représentées par des flèches ayant des *déclencheurs* qui testent la présence ou l'absence d'un signal, des *conditions* qui sont un prédicat utilisant la valeur des signaux et des *effets* représentants des actions instantanées) ne peuvent pas couper les limites du macroétat comme avec les STATECHARTS. La transition $e_1(c_1)$ de la Figure 1.6 montre un exemple où les limites de l'état hierarchique sont coupées.

Pour exprimer les communications et les synchronisations internes du modèle, les SYNCCHARTS permettent l'utilisation de signaux. Ces signaux peuvent être déclarés externes ou locaux à des macroétats. La Figure 1.8 montre un exemple d'un compteur 2-bits implémenté avec des SYNCCHARTS. On peut y voir deux bascules fonctionnant en parallèle dans le macroétat **bascules**. Il y a aussi un signal interne **C0** qui est émis par la première bascule après deux occurrences du signal d'entrée **T**. Après quatre occurrences de ce signal, le signal de sortie **C** sera émis par la deuxième bascule. Le compteur commence à compter lorsque le signal **start** est activé et arrête lorsque le signal **stop** l'est.

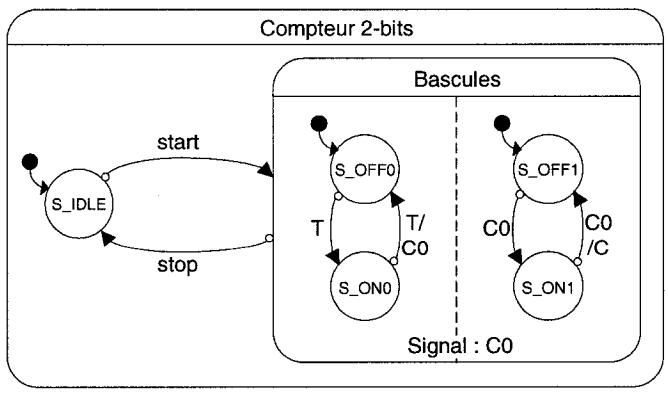


Figure 1.8 Exemple d'un compteur binaire 2-bits à l'aide de SYNCCHARTS

Les SYNCCHARTS sont exécutés « cycliquement » comme tous les autres modèles synchrones. Un cycle est instantané et comprend trois phases : *lecture des entrées, calcul de la réaction, mise à jour des sorties.* Chacune des réactions du modèle est déterministe et les réactions successives, qui correspondent à des instants successifs, ne se chevauchent pas. Pour plus d'informations, référer à [1] et [2].

Esterel-Technologies a utilisé les SYNCCHARTS dans ses outils Esterel Studio (Section 1.5.1 et [3]) et SCADE Suite (Section 2). Dans ce dernier, le nom de SSM (*Safe State Machines*) leur a été donné.

1.3.3 STATEFLOW

Le langage STATEFLOW est un langage de spécification graphique qui combine des machines à états finis hiérarchiques dans le même genre que STATECHARTS (par contre, leur sémantique est complètement différente) et les diagrammes de flux (*flowcharts*). STATEFLOW est utilisé dans la suite de logiciel Matlab Simulink de Mathworks (voir section 1.5.2) pour permettre la modélisation de systèmes « par-évènements ».

Un modèle STATEFLOW possède toujours un état actif. L'exécution d'un modèle consiste à exécuter l'état actif chaque fois qu'un évènement se produit dans l'environnement. Les évènements peuvent venir d'une horloge ou d'entrées du système [31]. Plus précisément, l'exécution d'un modèle se déroule comme suit :

1. Vérifier si une transition doit être prise, dans l'affirmative la prendre sinon passer à 2
2. Exécuter les actions internes (y compris les transitions internes)
3. Exécuter tous les états internes qui sont actifs

Tout comme STATECHARTS, le langage permet des transitions conditionnelles, de ses transitions interniveau, des états s'exécutant en parallèle, de ses *broadcasts.* De plus,

STATEFLOW permet des opérations plus complexes comme les jonctions de transitions. Pour une description complète du langage, référer à [38]

Le principal problème de STATEFLOW est l'absence d'un formalisme du langage. Ceci devient un problème lorsque STATEFLOW doit être utilisé dans des projets de systèmes critiques. Beaucoup de recherches ont été faites pour essayer de formaliser le langage en essayant entre autres de ne permettre seulement l'utilisation d'une sous-partie du langage [14], [27]. D'autres ont essayé de spécifier une sémantique opérationnelle formelle, comme [31].

1.4 Langages de spécification

Tous les langages vus précédemment peuvent être considérés comme des langages de spécification. C'est à dire des langages à partir desquels un système sera spécifié. Dans l'idéologie « Orienté-modèle », les modèles créés à partir de ces langages peuvent même être exécutés et vérifiés avant l'implémentation. Par la suite, du code peut-être généré directement à partir de cette spécification. Cette section présentera deux langages dits de spécification.

1.4.1 RSML

RSML (Requirements State Machine Language) est un langage de spécification graphique basé sur les STATECHARTS. En effet, lorsqu'utilisées pour décrire des spécifications, certaines constructions faites avec STATECHARTS peuvent être difficiles à comprendre. Le langage RSML a été créé dans l'optique de faciliter la compréhension des spécifications. Pour ce faire, RSML emploie seulement certaines des possibilités de STATECHARTS et en définit de nouvelles. Par exemple, RSML utilise les concepts d'états hiérarchiques, de la décomposition ET, des tableaux et des

transitions conditionnelles empruntés à STATECHARTS. Les opérateurs historiques ne sont pas supportés, la communication est faite par des canaux au lieu des *broadcasts* [19]. RSML donne la possibilité d'ajouter au design une description des interfaces et la notation syntaxique a été modifiée. Ces deux derniers ajouts ont été faits pour faciliter la lecture et les revues des spécifications. Pour plus d'informations sur le langage, référer à [37].

1.4.2 ADORA

ADORA (Analysis and Description Of Requirements and Architecture) est un langage de spécification graphique basé sur UML (Unified Modeling Language) [10]. L'idée principale d'ADORA est de modéliser les aspects comportement, fonctionnalité et données dans un seul objet d'un environnement hiérarchique. La décomposition hierarchique est la fonctionnalité clé d'ADORA dans laquelle la sémantique pour exprimer les comportements est basée à partir des STATECHARTS [6]. De plus, contrairement à UML, ADORA utilise des objets abstraits au lieu des classes. Ceci permet de donner la carte complète de l'intéraction entre les modules, en instanciant plusieurs de ces objets. Par exemple, on peut avoir dans un design un objet de type « gestion de la température » qui pourrait être utilisé par plusieurs modules. Dans le design ADORA, on instancierait un objet « gestion de la température » dans chacun des modules qui doit l'utiliser [57]. Pour plus d'informations, référer à [6] et [57].

1.5 Outils

Cette section présentera les principaux outils de modélisation dans le domaine des langages synchrones et des systèmes réactifs. Ces outils sont considérés comme étant « Orienté-modèle ».

1.5.1 Esterel Studio

Esterel Studio est un logiciel de modélisation d'Esterel-Technologies basé sur le langage ESTEREL. Celui-ci est spécialisé dans le design de composants matériels comme les DMA, les protocoles, les contrôleurs de câche, les sous-systèmes E/S, etc. En particulier, un design Esterel Studio respecte le formalisme et le déterminisme du langage ESTEREL. Ceci permet la vérification formelle de propriétés du modèle à l'aide de l'outil d'analyse formelle et d'ainsi avoir des designs de plus grande qualité.

La modélisation s'effectue avec un mélange d'utilisation des SYNCCHARTS (voir section 1.3.2), et du langage ESTEREL. Ceci permet de spécifier le chemin de données (équations et arithmétique) à l'aide d'ESTEREL et d'exprimer graphiquement le comportement du modèle à l'aide des SYNCCHARTS. De plus, cette notation est synthétisable à 100%. Pour plus d'informations, référer à [25].

1.5.1.1 Génération de code

Esterel Studio permet la génération automatique de code synthétisable RTL (VHDL et Verilog) en plus de la génération de code C/C++ et SystemC. En effet, la maxime d'Esterel Studio est « *What you specify is what you synthesize* » (ce que vous spécifiez est ce que vous synthétisez). C'est-à-dire que la génération de code fera en sorte que le comportement du modèle spécifié sera exactement le même que celui du code généré et synthétisé. Cette qualité est très importante dans le cas de design de systèmes critiques.

1.5.2 Matlab Simulink

Matlab Simulink est un outil de modélisation graphique de la firme Mathworks. Celui-ci est principalement spécialisé dans le prototypage avec lequel il est très facile et rapide de représenter graphiquement des équations numériques et de contrôle pour ensuite les simuler. Il est possible d'utiliser l'outil pour faire de la spécification, du design, de l'implémentation et des tests et de la vérification.

Les STATEFLOW permettent la modélisation avec Simulink de systèmes réactifs « par évènements ». De plus, il est possible d'incorporer des équations et des algorithmes Matlab aux designs, ainsi que du code C, Fortran et ADA. Pour plus d'informations sur l'outil, référer à [53].

Un des principaux problèmes de Matlab Simulink lors de son utilisation pour la modélisation de systèmes critiques est la présence de constructions ou d'opérateurs créant des ambiguïtés, sapant ainsi le déterminisme [16]. En effet, contrairement à des outils comme SCADE Suite et Esterel Studio ou le déterminisme a toujours été un facteur très important dès leur conception, Matlab Simulink mise sur une plus grande flexibilité. Ceci a comme inconvénient majeur d'affecter le déterminisme des modèles.

1.5.2.1 Génération de code

Il est possible de faire de la génération de code à partir d'un modèle Matlab Simulink. À partir de l'outil Real-Time Workshop de Mathworks, du code ANSI/C peut être généré et ainsi être utilisé sur la plupart des processeurs et avec la plupart des systèmes d'exploitation temps-réel. L'outil SMT6040 de Sundance Multiprocessor Technology Ltd. permet de générer du code VHDL à partir d'un modèle Simulink [50].

1.5.3 Rhapsody

Rhapsody est une suite de développement graphique « Orienté-modèle » de la firme I-Logix basé sur UML 2.0. En effet, Rhasody utilise un diaclecte UML appelé « Real-Time UML ». Les propriétés de ce dialecte en font une excellente variante des STATECHARTS pour modéliser le comportement et les intéractions avec le monde extérieur [8]. En effet, on utilise Rhapsody pour modéliser hiérarchiquement des systèmes réactifs temps-réel avec des diagrammes UML et des machines à états. Une fois le modèle complété, il est possible de le tester et de le valider. Rhapsody offre un outil de génération de code permettant de générer du C, C++ et de l'ADA. Pour plus d'informations, référer à [52].

1.5.4 Autres outils

D'autres outils pour la modélisation de systèmes réactifs sont aussi offerts sur le marché. *Polychrony* [22] est un outil de modélisation basé sur le langage SIGNAL. *PtomelyII* [21] est un outil graphique permettant de modéliser des systèmes réactifs. *Rational Rose/RT* [34] est un outil utilisé pour implémenter du UML temps-réel.

1.6 Certification

L'industrie avionique civile oblige que tout logiciel critique pour la sécurité doit être certifié par une autorité de certification avant tout usage dans un avion commercial. Les principales autorités de certification sont la FAA (Federal Aviation Administration) aux États-Unis, la JAA (Joint Aviation Authorities) en Europe et Transport Canada au Canada. La norme DO-178B détermine les objectifs du processus logiciel que les compagnies qui développent des équipements devront

respecter et que les autorités devront vérifier pour permettre la certification.

1.6.1 DO-178B

Le document « *Software Considerations in Airborne Systems and Equipment Certification* » **RTCA/DO-178B** [46] décrit la norme DO-178B qui doit être suivie pour permettre la certification logicielle. Le principal objectif de la norme est d'assurer que les logiciels accomplissent les fonctions pour lesquelles ils ont été prévus (avec un certain niveau d'assurance). En particulier, on y décrit les objectifs des processus de cycle de vie des logiciels, les activités et les considérations de design permettant de réaliser ces objectifs, et la description des preuves montrant que les objectifs ont été satisfaits. Cinq niveaux d'assurance des logiciels sont définis dans la norme. Ces niveaux d'assurance indiquent jusqu'à quel point un logiciel est critique pour le bon fonctionnement de l'avion et ce qui pourrait arriver lors d'une défaillance.

- **Niveau A**: Défaillance catastrophique pour l'aéronef (ex.: mène à l'écrasement)
- **Niveau B**: Défaillance sévère pour l'aéronef (ex.: des gens pourraient être blessés)
- **Niveau C**: Défaillance majeure pour l'aéronef (e.x: le système de vol ne fonctionne plus et le pilote doit piloter en mode manuel)
- **Niveau D**: Défaillance mineure pour l'aéronef (ex.: problèmes de communication entre le pilote et les contrôleurs aériens)
- **Niveau E**: Aucun effet sur l'aéronef (ex.: le système de divertissement est en panne)

Techniquement, la norme spécifie que chaque ligne de code est directement traçable jusqu'à la spécification et jusqu'à un test. De plus, aucun code « orphelin » ne doit être inclus dans le programme final. Pour plus d'informations, référer à [26] et [23].

CHAPITRE 2

L'OUTIL SCADE

SCADE (**S**afety **C**ritical **A**pplication **D**evelopment **E**nvironnment) est un outil de la firme Esterel-Technologies qui se spécialise dans le développement d'applications embarquées critiques pour la sécurité. Ce type d'application est généralement utilisé dans le domaine de l'industrie de l'avionique, du nucléaire et depuis quelques années dans l'automobile. Tous les aspects couverts et toutes les figures utilisées dans ce chapitre ont été pris de la version 5.1 de la suite logicielle SCADE Suite.

2.1 Idéologie

Esterel-Technologies définit SCADE comme un outil permettant d'automatiser la production de logiciels embarqués à partir de leur description formelle sous forme de modèle. En effet, SCADE est très axé sur la qualité du design tout en ne sacrifiant pas la productivité. Le modèle de développement basé sur l'idéologie « Orienté-Modèle » permet d'utiliser SCADE pour faire la spécification jusqu'au code à être implanté sur la cible. SCADE peut être considéré comme étant une méthodologie complète, c'est-à-dire basée sur une sémantique rigoureuse, sur un outil de génération de code et sur un outil de test et de vérification. Le formalisme de SCADE permet au designer de modéliser ce qu'il pense et permet à tout le monde de le comprendre [25].

2.2 Concepts

Cette section présente des concepts sur les langages utilisés par SCADE.

2.2.1 Propriétés du langage

SCADE utilise deux modèles formels de spécification. Le premier, les diagrammes bloc, est utilisé pour spécifier le flux de données d'une application embarquée. Le langage SCADE qui est utilisé par les diagrammes bloc est basé sur le langage synchrone LUSTRE. Le deuxième, les machines à états sécuritaires (Safe State Machines), est utilisé pour spécifier le flux de commande. Les SSM sont basées sur le langage graphique SYNCCHARTS.

2.2.1.1 Utilisation de LUSTRE

Le langage SCADE est basé sur une représentation graphique de LUSTRE, les diagrammes blocs, pour modéliser le flux de données. Ceci permet à SCADE d'avoir un langage de description fortement typé, structuré, hiérarchique et ayant une sémantique déterministe. Les modélisations SCADE ne permettent aucune boucle interne, aucune allocation dynamique. De plus, le temps d'exécution maximum peut continuellement être borné. Pour ce faire, SCADE exécute toujours la partie vraie et la partie fausse d'une condition if-else, et fait la mise à jour des variables impliquées par la suite. En fait, le temps d'exécution est toujours le même. Ceci permet à SCADE de traiter des problèmes de manière formelle autant pour les résultats que pour le temps d'exécution.

2.2.1.2 Utilisation des SYNCCHARTS

Pour modéliser le flux de commande, SCADE utilise les SSM, basées sur le langage graphique SYNCCHARTS. Les SSM sont hiérarchiques, concurrentes et communiquent entre elles à l'aide de signaux. Celles-ci permettent l'utilisation de machines à états simples et de macroétats (voir section 1.3.2). La sémantique des SSM ne

permet aucune construction non sécuritaire qui pourrait être tolérée dans d'autres langages graphiques. Par exemple, à cause de l'hypothèse synchrone qui affirme qu'une transition doit être instantanée, des transitions qui dépassent les frontières des macroétats et des transitions prises à moitié ne sont pas permises (à la fin d'un cycle, la SSM ne sera jamais arrêtée « entre deux états » mais bien toujours dans un état précis) [7]. Toute construction qui met en cause le déterminisme sera ainsi rejetée à la compilation ou lors de la conception (l'éditeur SCADE interdisant certaines constructions problèmatiques).

2.3 Environnement graphique

Cette section décrira les constructions possibles à l'aide de l'environnement graphique de SCADE.

2.3.1 Noeuds

Une modélisation graphique faite avec SCADE est composée de noeuds. Un noeud peut être vu comme une boîte noire qui peut être accédée seulement par son interface composée d'entrées (représentées par le symbole $\mapsto$) et de sorties (représentées par le symbole $\rightarrow$). Un programme SCADE sera vu hiérarchiquement comme un noeud principal composé de noeuds communiquant entre eux (voir Figure 2.1 où le *Noeud_Principal* se décompose en deux noeuds *CalculVoltage* et *CalculPuissance*). Chaque noeud de SCADE peut être de nature différente, soit : **diagramme bloc**, **SSM** ou **code importé**.

Pour permettre une vue d'ensemble plus rapide, un noeud n'a pas besoin d'être complètement défini avant qu'on puisse l'utiliser dans le design. De plus, ceux-ci sont complètement modulaires, c'est-à-dire que leur comportement ne dépend pas de

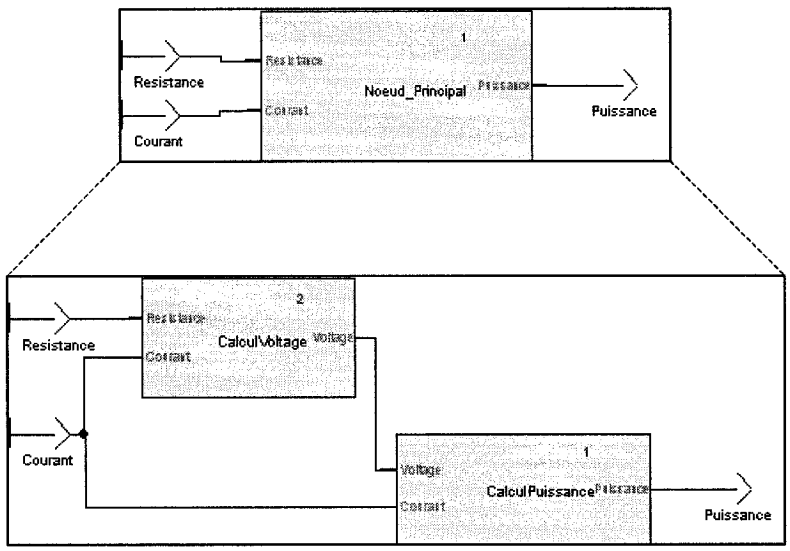


Figure 2.1 Exemple d'une hiérarchie de noeuds dans un programme SCADE

leur contexte. Une séquence d'entrée donnera toujours la même séquence à la sortie. Avec SCADE, il est très facile d'utiliser et de naviguer dans les noeuds. Un simple clique nous permet de voir l'intérieur (c.-à-d. l'implémentation) de ceux-ci.

2.3.2 Diagrammes bloc

Les diagrammes bloc sont utilisés pour modéliser le flux de données. Les noeuds diagrammes bloc font des calculs mathématiques, implémentent des filtres et des délais. Les données du flux de sortie sont calculées à partir du flux d'entrée. Tous les blocs sont concurrents et ne peuvent communiquer que par leurs entrées et leurs sorties. Les blocs peuvent être conditionnels (« condact »), c'est-à-dire activés seulement lorsque la condition d'activation est vraie. La Figure 2.2 donne un exemple d'un diagramme bloc composé d'une équation simple (noeud *CalculVoltage*) et d'un bloc d'activation conditionnel (noeud *CalculPuissance*) qui est exécuté lorsque l'entrée booléenne *Activation* est vraie. La valeur 0.0 attribuée au noeud conditionnel *CalculPuissance* sert de valeur initiale à la sortie *Puissance*.

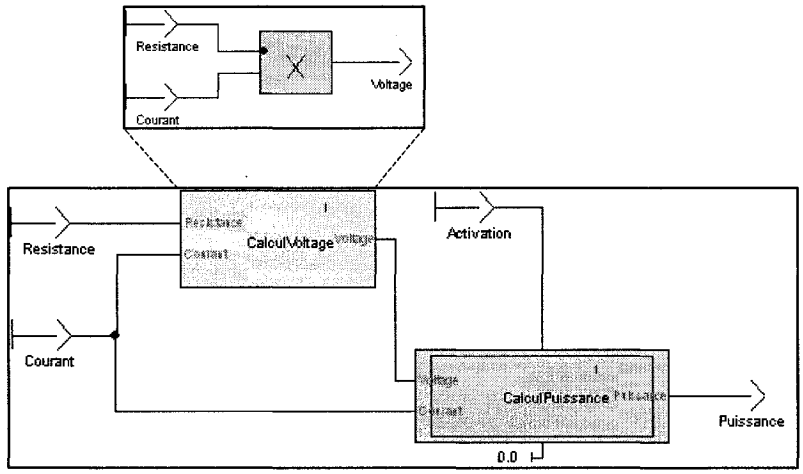


Figure 2.2 Exemple d'un diagramme bloc dans SCADE

Le support des blocs hiérarchiques permet de maintenir un ensemble de blocs de base minimal. Sémantiquement, il est mieux de définir des blocs hiérarchiquement à partir de blocs plus simples que d'écrire des fonctions complexes. Ceci améliore la lisibilité tout en ne sacrifiant pas la performance. Cette hiérarchie, purement architecturale, a été gardée le plus simple possible en évitant toutes règles complexes [7].

2.3.3 Safe State Machines

Les machines à états sécuritaires (SSM) sont utilisées par SCADE pour modéliser le flux de commande. Une condition d'activation peut être associée à chacune des transitions. Dans ce cas, les variables impliquées dans la condition deviennent des entrées du bloc SSM. Les SSM offrent la possibilité d'émission d'un signal lorsqu'une transition est prise, lorsqu'on entre dans un état ou lorsqu'on en sort. Dans ces cas, les signaux émis (par des variables) deviennent des sorties du bloc SSM. Il est aussi possible d'utiliser des signaux internes pour faire communiquer des machines à états concurrentes. Dans tous ces cas, des relations mathématiques peuvent être employées pour manipuler les variables ou les signaux.

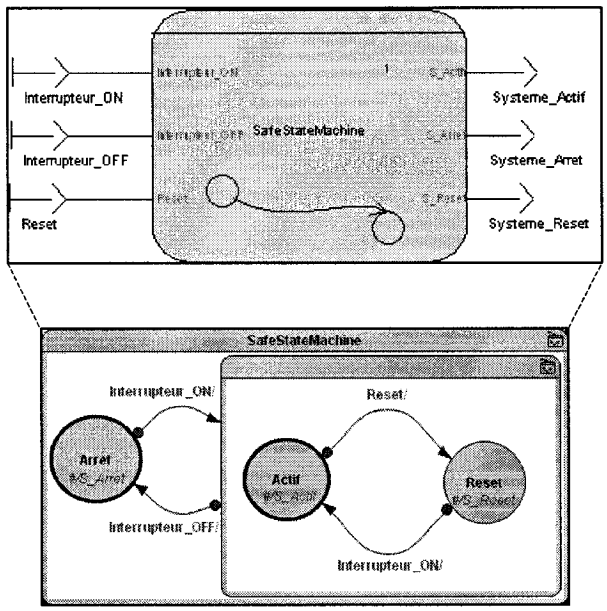


Figure 2.3 Exemple d'un SSM dans SCADE

La Figure 2.3 montre un exemple de SSM où chaque transition est associée à une condition booléenne (la transition est prise lorsque la variable est vraie) et où chaque état émet un signal lorsque la transition est activée (variables *S_Arret*, *S_Actif*, *S_Reset*). Ces conditions booléennes deviennent des entrées du bloc SSM et les signaux émis les sorties.

Les SSM peuvent être par la suite utilisées dans un diagramme bloc, où les signaux émis en entrée et en sortie deviennent de simple booléens. Cette modélisation mixte est souvent utilisée pour définir un mode de fonctionnement de la partie flux de données (par exemple : mode arrêt, mode actif, etc.).

2.3.4 Code importé

SCADE offre la possibilité de définir des noeuds à partir de code importé. Cette fonctionnalité permet à un design de s'interfacer à son environnement ou simplement réaliser des descriptions qui n'auraient pas été possibles avec le langage SCADE.

Pour ce faire, il suffit d'ajouter un noeud « code importé » dans le modèle et de lui définir une interface (entrées et sorties). Le code sera défini dans un fichier externe et associé au noeud. Le code importé peut seulement être écrit en C ou en ADA, selon du générateur de code utilisé par la suite (voir Section 2.4 sur la génération de code).

2.3.5 Analyseur de design

L'analyseur de design est analogue au compilateur (en particulier lors des étapes de l'analyse de la syntaxe et de la sémantique) dans les langages de programmation. En effet, on s'en sert entre autres pour vérifier qu'aucune erreur de syntaxe ou de sémantique ne s'est introduite dans la modélisation. La vérification de la syntaxe s'assurera que chaque entrée et chaque sortie sont branchées et que toutes les entrées et les variables locales sont consommées. L'analyse de la sémantique ne vérifiera qu'aucune récursion et qu'aucune boucle infinie ne sont présentes dans le design. De plus, l'analyseur s'assurera que tous les types sont bien utilisés. Le langage SCADE étant fortement typé, il est interdit, par exemple, de brancher directement un signal de type entier avec un autre de type réel.

2.3.6 Simulation

SCADE offre la possibilité de simuler directement le modèle. SCADE garantit que le comportement à la simulation sera exactement le même que le comportement du code généré avec la même configuration (voir section 2.4 pour plus de détails). La simulation est modulaire au niveau noeud, elle est synchrone (on change les entrées, on fait un pas d'exécution, on lit les sorties) et déterministe. Il est possible d'y faire une simulation pas à pas, d'un certain nombre de cycles ou continue. La simulation peut être automatique (à l'aide de scénarios qui décrivent quelles valeurs les entrées doivent prendre à quels cycles) ou manuelle. Dans ce dernier cas, l'utilisateur change les

valeurs des entrées. Une interface usager écrite en JAVA peut aussi être associée à une simulation. Ainsi, un utilisateur pourrait changer des valeurs, peser sur des boutons, et voir le résultat directement sur l'interface. Ceci est réalisable, car SCADE permet de « brancher » les entrées et les sorties du design avec des variables d'un programme JAVA. Bien sûr, il est aussi possible d'ajouter des points d'arrêts (conditionnel ou non) et des assertions sur les noeuds. L'interface graphique du simulateur permet de voir la valeur de toutes les variables dans le design à chaque pas d'exécution et la possibilité de sauvegarder les résultats dans un fichier excel. De plus, une vue forme d'onde et une vue surveillance de variables sont aussi offertes.

2.4 Génération de code

À partir d'un design SCADE, du code source ADA ou ANSI C peut être généré. La Figure 2.4 présente les principales étapes de la génération de code où le modèle SCADE sera traduit dans un modèle LUSTRE qui sera lui-même traduit en code ADA ou en ANSI C. Ce code peut-être par la suite directement utilisé et compilé dans des projets. SCADE garantit une traçabilité complète du code jusqu'au modèle SCADE (grâce aux commentaires générés dans le code). De plus, le code aura exactement le même comportement pour une même configuration. Ceci est un des avantages majeurs de la génération de code avec SCADE. En effet, les tests étant effectués sur la spécification, il n'est plus nécessaire de faire des tests sur le code (voir Sections 2.4.2.1 et le Chapitre 3 sur la méthodologie SCADE). À même l'interface graphique de SCADE, il est possible d'effectuer la génération de code et de configurer le générateur. Les générateurs de code offerts sont KCG 4.2 et 5.1 (pour la génération ANSI C de code qualifiable), Spark et Standard ADA (pour la génération de code ADA). Il est aussi possible de configurer les optimisations faites par le générateur (pour plus de détails voir Section 2.4.3)

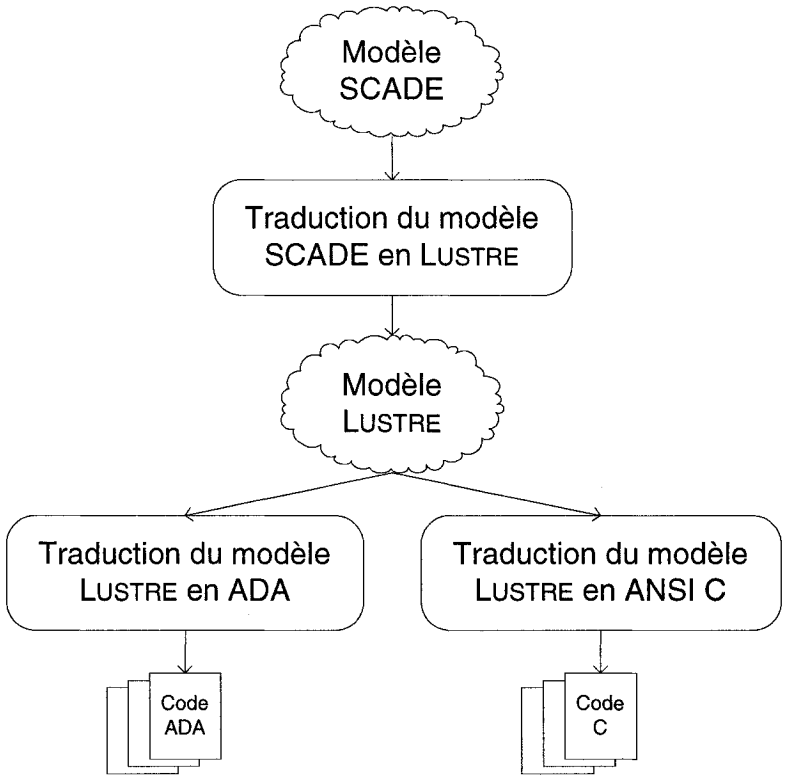


Figure 2.4 Étapes de la génération de code

2.4.1 Génération de SCADE à LUSTRE

La première étape de la génération est de traduire le modèle SCADE en modèle LUSTRE. Le générateur SCADE2LUSTRE est utilisé pour effectuer cette transcription. La description SCADE étant basée sur LUSTRE, la traduction est relativement directe. Par exemple, la syntaxe pour un noeud importé est :

- `imported node SQRT (x : real) returns (sqrt_x : real);`

et en LUSTRE :

- `function SQRT (x : real) returns (sqrt_x : real);`

Par contre, il existe certains concepts offerts par SCADE qui ne sont pas disponibles avec le langage LUSTRE. Le générateur doit alors transformer ces constructions syntaxiques en une construction équivalente en LUSTRE. Le Tableau 2.1 donne une

liste des principales différences et pour chacune d'elle, le type de transformation effectuée pour passer de SCADE à LUSTRE.

Tableau 2.1 Principales différences entre SCADE et LUSTRE

MODÈLE SCADE	Traduction LUSTRE
Fait de plusieurs fichiers	Traduit dans un seul fichier
Possède des types énumérés	Deviens un type **int** où chaque valeur devient une constante
Possède des types génériques utilisables dans des noeuds génériques	Versions différentes de chaque appel du noeud selon les types utilisés
Un noeud peut posséder des opérateurs cachés	Les opérateurs cachés deviennent des entrées du noeud
Constantes définies à partir d'autres constantes	Les constantes sont traduites en valeurs

La génération des SSM se fait différemment. En effet, la première étape est de traduire le modèle SSM en code ESTEREL (ceci vient du fait qu'ininitialement les SSM ont été développées pour le logiciel Esterel Studio). Par la suite, le compilateur STRL2LUSTRE sera utilisé pour traduire le code ESTEREL en code LUSTRE.

2.4.1.1 Ordonnancement statique du code

Une description LUSTRE peut être facilement traduite en code impératif séquentiel (comme C et ADA) à l'aide d'une simple boucle infinie comme à la Figure 2.5 [29]. La difficulté ici est de trouver le bon ordonnancement des calculs pour ne pas créer des problèmes de dépendance entre les variables. Généralement, une simple analyse topologique permet de trouver un bon ordre. Par définition, les programmes LUSTRE ne peuvent pas comporter de définitions cycliques syntaxiques. Ceci permet d'assurer une dépendance statique entre les flux (les programmes LUSTRE faisant toujours des opérations sur des flux, voir Section 1.2.1.2) et de permettre une génération très simple de code. De plus, SCADE ne permet pas qu'une sortie d'un noeud soit directement utilisée comme une entrée du même noeud sans qu'un opérateur « **pre** » ne soit utilisé

```
initialisations
début_boucle
        lecture entrées
        calcul des sorties
        mise à jour des mémoires
fin_boucle
```

Figure 2.5 Exemple d'une boucle infinie simple

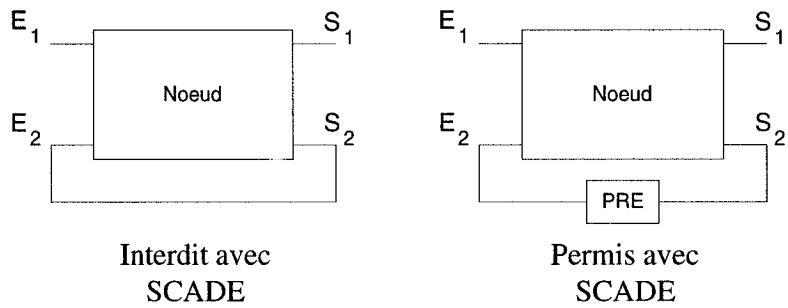


Figure 2.6 Boucle de rétroaction avec SCADE

(voir Figure 2.6). Ceci permet au générateur d'ordonnancer les noeuds séparément avant d'ordonnancer le système au complet, car l'ordre d'exécution des équations d'un noeud ne peut plus être affecté par la façon dont le noeud est appelé. Sans cette contrainte, le générateur devrait ordonnancer à la fois toutes les équations du programme, ce qui serait beaucoup plus compliqué et beaucoup plus long [4].

Avant d'ordonnancer les SSM, SCADE s'assurera toujours qu'un état d'équilibre peut être atteint par la machine à états. En effet, une machine à états doit toujours pouvoir s'arrêter dans un état stable. Lorsque cette condition est remplie, la transcription ESTEREL des SSM est transformée en équations booléennes ordonnancées.

2.4.2 Générateur de code source (ANSI C ou ADA)

La dernière étape de la génération est de traduire la description LUSTRE en un code impératif ANSI C ou ADA. En effet, cette traduction consiste à créer une fonction

d'initialisation et une ou plusieurs fonctions implémentant le cycle d'exécution sur lequel la fonction est basée. La traduction sera faite avec le générateur LUSTRE2C (pour le langage C) et LUSTRE2ADA (pour le langage ADA).

Avant de démarrer la génération de code, l'utilisateur doit indiquer à SCADE quel noeud de son modèle est le noeud principal. Ce noeud est généralement le noeud de plus haut niveau hiérarchique du design. Comme indiqué plus tôt, deux catégories de fonctions seront alors générées. La première catégorie, les fonctions d'initialisations, implémentent les valeurs d'initialisations des noeuds conditionnels et des variables initialisées à partir de l'opérateur « -> » (voir Section 1.2.1.2). La deuxième catégorie de fonctions générées décrit le fonctionnement des noeuds (celles-ci auront des noms semblables aux noms des noeuds).

Le générateur de code source placera l'implémentation des noeuds, la description des variables globales, des variables utilisateur et des constantes dans des fichiers distincts. De plus, un fichier « en-tête » sera généré pour chacun des noeuds importés. Ces fichiers devront être complètés par le designer pour y interfacer le code importé.

Une série de commentaires sera produite dans tous les fichiers sources générés. Ces commentaires sont utilisés pour augmenter la lisibilité du code et pour faciliter la traçabilité. En effet, les commentaires indiquent à quel noeud SCADE correspond le code. Les commentaires ajoutés manuellement au design SCADE sont aussi inclus dans le code source.

2.4.2.1 Générateur de code qualifiable (KCG)

SCADE possède un générateur de code ANSI C qui est qualifié. Au sens de la DO-178B, un outil qualifié est un outil qui opère une transformation d'un format vers un autre de façon connue et démontrée. Donc, un générateur de code qualifié est

un outil pour lequel il a été démontré (par des tests, des relectures du code ou des preuves formelles) que la transformation de code qu'il effectue fonctionne comme elle a été spécifiée. En d'autres termes, des lois de correspondances sont écrites entre deux langages et il est démontré que ces lois sont inviolables. Ainsi, le comportement de l'outil devient qualifié.

Le générateur KCG produit du code qui est qualifiable au niveau A (le niveau le plus haut) de la DO-178B (voir 1.6.1). Le code est considéré « correct-by-construction » étant donné que le générateur de code à lui-même été qualifié par le même processus que l'application [7].

Lorsque les autorités de certification décident de certifier un logiciel au sens de la DO-178B, ils approuvent en fait le processus de développement qui a mené à la conception du logiciel. L'utilisation d'outils qualifiés dans un processus de conception permet d'accélérer celui-ci. Aux yeux des autorités, une série d'activités remplacée par l'outil ne devront plus être examinées, et ce, tout en étant assurés que ce remplacement ne cause aucun problème. Le chapitre 3 sur la méthodologie SCADE donne plus de détails sur ces aspects.

2.4.3 Optimisations du code

La génération de code peut être configurée pour effectuer des optimisations sur le code ADA ou ANSI C. À partir de l'interface graphique de SCADE, les optimisations suivantes sont disponibles :

- *Par défault* :
    1. Toutes les variables internes qui ne sont pas utilisées lors du calcul des sorties sont considérées comme étant du « code inutilisé » et sont éliminées.
    2. Génération prédéfinie des opérateurs « pre » et « fby ».

3. Les variables locales inutiles (c.-à-d. toujours égale à une entrée, une constante, etc.) sont éliminées.
4. Les comparaisons entre expressions structurées sont traduites en une fonction de comparaison de la mémoire, offrant ainsi un gain de vitesse (disponible seulement en C).

- *Constantes* : En C, les constantes seront définies avec le mot-clé **#define**, au lieu d'être de simples variables
- *Variables usagers* : Empêche l'optimisation des variables créées par le designer dans le modèle SCADE
- *Variables internes* : Élimine les variables internes générées par SCADE.

D'autres optimisations sont aussi disponibles lors de la génération, par exemple on peut choisir d'utiliser des opérateurs de manipulation de bits, de protéger les constantes (seulement en C où les constantes seront générés avec le mot clé **const** et seront ainsi placées dans la zone protégée de la mémoire), d'ignorer les assertions, de grouper les variables de même type (locales, globales) dans une même structure ou de générer un fichier source (C ou ADA) par noeud au lieu d'un seul fichier source (C ou ADA) pour tout le projet.

2.4.4 Générations particulières

Il est possible de générer le code de façon à pouvoir l'utiliser directement dans des systèmes précis. SCADE permet de générer du code à être utilisé dans les systèmes d'exploitation temps-réel OSEK [41] et μC/OS-II [40]. Il est aussi possible de générer du code à être utilisé dans Matlab Simulink (voir Section 1.5.2). SCADE génèrera alors des fichiers adaptateurs qui pourront être inclus dans ces projets particuliers. Ces fichiers adaptateurs feront alors des appels aux fichiers de code source générés par SCADE.

CHAPITRE 3

LA MÉTHODOLOGIE SCADE

Ce chapitre introduit la méthodologie utilisée dans l'industrie et celle proposée par SCADE pour le développement d'applications critiques. De plus, les résultats d'une expérience d'utilisation de l'outil SCADE menée avec des étudiants gradués y sont présentés.

3.1 Un défi industriel

Le développement d'applications critiques pour la sécurité en avionique, dans le domaine de l'automobile, dans le nucléaire, etc., présente tout un défi pour les industries impliquées. En effet dans ce genre de systèmes, la sécurité n'est pas optionnelle, mais bien un requis absolu. C'est la raison pour laquelle le développement de ce type de logiciel présente des coûts très élevés. Dans le domaine de l'avionique, il a été observé que [15] :

- Le temps moyen de développement et de test de 10000 lignes de code suivant un niveau de sécurité B de la DO-178B est de 16 années-personnes.
- En moyenne, un projet en avionique coûte 46% de plus que le budget initial et dépasse de 35% les échéanciers.
- Le temps de mise sur le marché est de 3-4 années.
- Le coût d'une défaillance logicielle mineure est d'environ 100 000$ à 500 000$.
- Le coût d'une défaillance logicielle majeure est d'environ 1 à 500 millions $.

En considérant tous ces coûts, toute amélioration menant à une accélération du processus de développement de logiciels critiques qui ne sacrifie rien de la sécurité peut devenir très intéressante pour les industries impliquées dans de tels projets de conception. En particulier celles permettant de diminuer les coûts du projet.

3.2 Le développement d'applications critiques en industrie

Dans le domaine de l'avionique, toute industrie qui désire développer des applications critiques pour la sécurité doit suivre le processus logiciel de la norme DO-178B (voir Section 1.6.1). En suivant la DO-178B, les concepteurs fixent des objectifs à atteindre et par la suite vérifient qu'ils ont été réalisés. Ces objectifs décrivent généralement le niveau d'assurance de développement de l'application et les standards de conceptions. La vérification implique des revues de conceptions et des tests d'intégration qui assure que les objectifs ont été atteints. Par la suite, des procédures sont rédigées pour s'assurer que la vérification s'est bien déroulée. De telles procédures démontrent par exemple que les remarques découlant de la revue de conception ont été prises en considération et que la couverture structurelle du code a été réalisée.

3.2.1 Processus de développement

Le processus de développement logiciel recommandé par la DO-178B comprend quatre étapes principales (voir Figure 3.1).

La première étape, *les spécifications logicielles*, est celle où les spécifications de haut-niveau de l'application sont produites. Ces spécifications de haut-niveau décrivent les requis fonctionnels et opérationnels, les contraintes de temps et de mémoire, les interfaces logicielles et matérielles, les requis concernant la détection de défaillances et du monitorage de la sécurité.

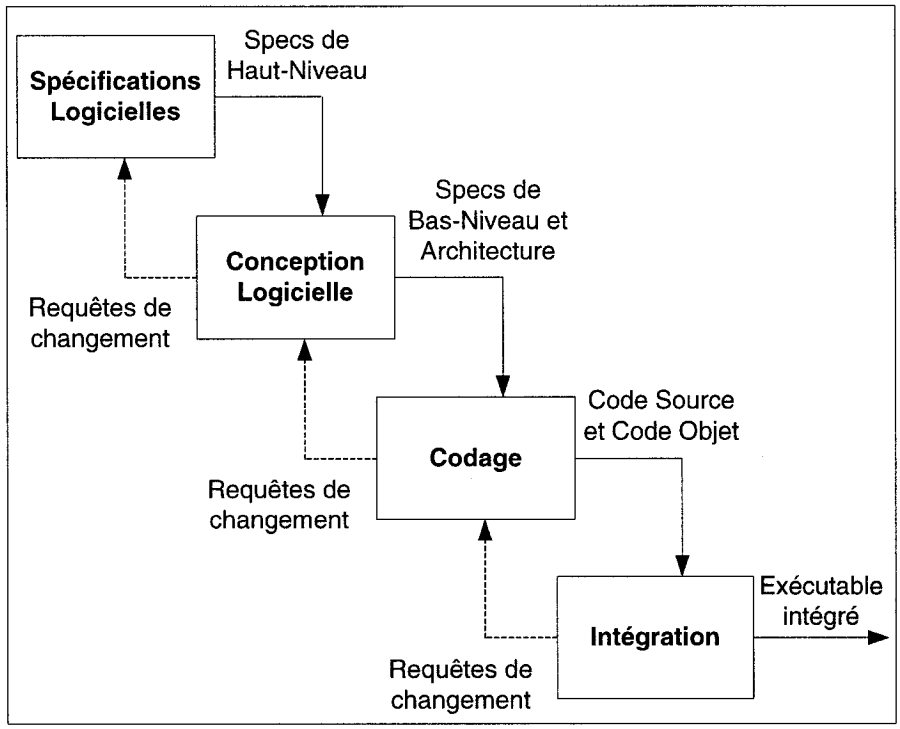


Figure 3.1 Processus de développement logiciel de la DO-178B

La deuxième étape, *la conception logicielle*, est celle où sont produites les spécifications de bas-niveau de l'application et son architecture. Les spécifications de bas-niveau sont réalisées à partir des spécifications de haut-niveau. Celles-ci décriront toutes les entrées et les sorties du système, le flux de données et de commande, les limitations des ressources, l'ordonnancement de l'application, les mécanismes de communication, et les composants logiciels.

La troisième étape, *le codage logiciel*, est celle où le code source de l'application est créé. Ce code source est fait directement à partir des spécifications de bas-niveau et doit être vérifiable et concis.

La quatrième étape, *l'intégration logicielle*, est celle où le code objet est obtenu, à partir du code source, et intégré à l'environnement.

À toutes les étapes, la traçabilité est requise. C'est-à-dire que les spécifications de

haut-niveau doivent être traçables jusqu'aux spécifications de bas-niveau et vice-versa, les spécifications de bas-niveau jusqu'au code source et vice-versa. Les « requêtes de changements » peuvent survenir lorsqu'un problème ou une erreur logicielle est détecté. Dans ce cas, une requête de changement sera émise, et le problème corrigé.

3.2.2 Processus de vérification

L'objectif principal du processus de vérification recommandé par la DO-178B est de « détecter et de faire un compte rendu des erreurs qui ont pu être introduites durant le processus de développement logiciel ». Corriger les erreurs est une activité du processus de développement logiciel et non du processus de vérification.

La vérification n'est pas seulement constituée de tests. En général, le test ne peut pas prouver l'absence d'erreur. Le processus de vérification implique aussi des revues (qualitatives et faites une fois) et des analyses (détaillées et reproductibles).

Le processus de vérification couvre toutes les étapes du processus de développement logiciel et vérifie les objectifs principaux suivants:

1. Les spécifications de haut-niveau ont été développées à partir des requis du système et celles-ci satisfont ces requis.

2. Les spécifications de bas-niveau ont été développées à partir des spécifications de haut-niveau et satisfont celles-ci.

3. Le code source a été développé à partir des spécifications de bas-niveau et satisfait celles-ci.

4. Le code objet exécutable satisfait les requis de l'application.

5. Le moyen utilisé pour démontrer la satisfaction des objectifs est techniquement correct et complet pour le niveau d'assurance désiré (voir niveaux: A, B, C, D,

E de la section 1.6.1).

La « satisfaction » des objectifs implique plusieurs éléments pour chacune des étapes du processus de développement. Par exemple, pour les spécifications de haut-niveau, il sera vérifié que chacune des spécifications est non ambiguë, suffisamment détaillée et n'entrant pas en conflit avec d'autres spécifications. Pour les spécifications de bas-niveau, il sera vérifié que celles-ci sont compatibles avec le système cible (du point vue de l'utilisation des ressources, des interfaces de celui-ci, des temps de réponse, etc.). Pour le code source, il sera vérifié qu'aucun code n'implémente des fonctions qui ne sont pas documentées. Pour plus de détails sur les différents points à vérifier pour chacune des étapes du développement logiciel, veuillez vous référer à la DO-178B Section 6 [46].

3.2.2.1 Les tests logiciels

Les deux principaux objectifs des tests logiciels sont de démontrer que l'application satisfait les requis et de démontrer avec un haut niveau de confiance qu'une erreur pouvant mener à une condition catastrophique a été éliminée.

Il existe trois types d'activités de tests:

1. Tests d'intégration matériel/logiciel qui vérifient que l'application s'exécute comme elle se doit dans son environnement cible.

2. Tests d'intégration logiciels qui vérifient les interrelations entre les spécifications systèmes et les composants.

3. Tests de bas-niveau qui vérifient l'implémentation des spécifications de bas-niveau.

Pour que ces tests soient satisfaisants :

- Ils devraient être basés sur les spécifications (et non sur le code source).
- Ils devraient être développés pour vérifier que les fonctionnalités sont correctes et des conditions qui pourraient mener à des erreurs devraient être déterminées.
- Des tests de couverture des spécifications devraient déterminer quelles spécifications n'ont pas été testées.
- Des tests de couverture structurelle devraient déterminer quelles structures logicielles n'ont pas été testées.

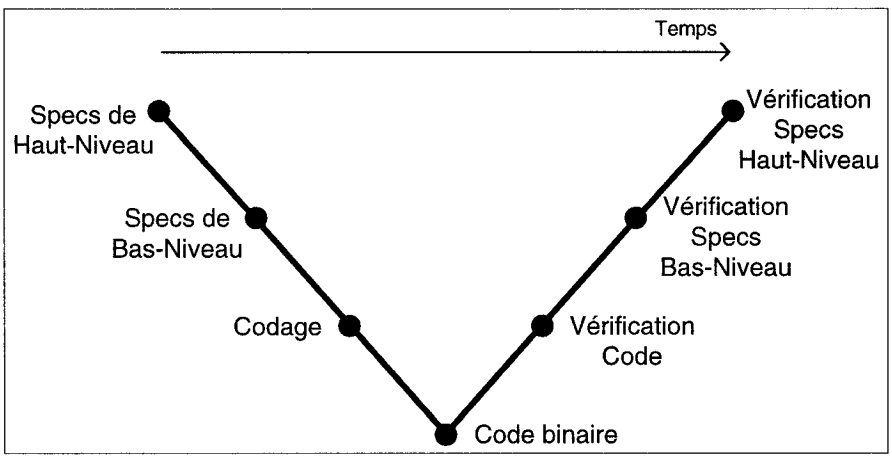


Figure 3.2 Cycle en V

3.2.3 Cycle en V

Le processus de développement logiciel et le processus de vérification peuvent être vus comme un cycle de développement ayant la forme d'un V (voir Figure 3.2). Le cycle en V représente l'évolution dans le temps du cycle de développement d'une application critique. Au départ (en haut à gauche du « V »), les spécifications haut-niveau sont écrites. Par la suite, à partir des spécifications haut-niveau, les spécifications de bas-niveau sont créées. À leur tour, ces spécifications de bas-niveau sont transformées en code source. À la pointe inférieure du cycle en V le

code source est compilé en code binaire et c'est aussi à ce moment que commence la vérification. En premier lieu, le code est vérifié et testé (à partir du code exécutable). Ensuite, les spécifications de bas-niveau sont aussi vérifiées et testées, et une analyse de correspondance directe entre le code et les spécifications de bas-niveau est faite (aucune fonctionnalité non documentée ou non implantée n'est tolérée). À la fin du cycle en V, les spécifications de haut-niveau sont testées et vérifiées et une analyse de correspondance entre celles-ci et les spécifications de bas-niveau est aussi effectuée. Après la fin du cycle, l'intégration et la validation de l'application (dans une situation réelle de fonctionnement) sont accomplies.

3.3 Le développement d'applications critiques avec SCADE

La norme DO-178B n'impose pas de choix d'outils ou de méthodologies pour implanter son processus de développement d'applications critiques. En effet, tant que les objectifs de la DO-178B sont respectés, n'importe quel outil peut être utilisé. SCADE propose une méthodologie « orienté-modèle » pour le développement d'applications critiques. Cette méthodologie permet, à l'aide de l'environnement de modélisation, de simulation, de vérification et de génération de code, d'utiliser les forces de SCADE et de son langage de modélisation pour accélérer le processus de développement classique d'applications critiques et diminuer ainsi les coûts.

3.3.1 Le flux de conception avec SCADE

L'environnement et le langage SCADE sont extrêmement bien adaptés pour l'expression de spécifications. En effet, à partir des spécifications systèmes, il est possible d'exprimer une certaine partie des spécifications de haut-niveau et la plupart des spécifications de bas-niveau à l'aide de SCADE. De plus, les spécifications de

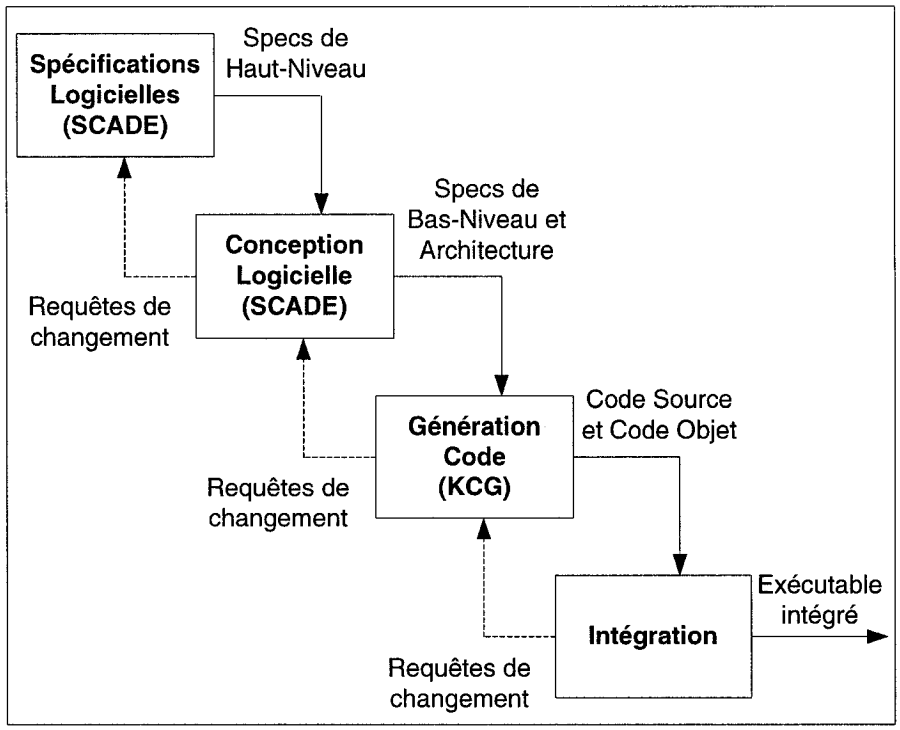


Figure 3.3 Processus de développement logiciel avec SCADE

haut-niveau ayant pu être exprimées dans SCADE n'ont pas besoin d'être traduites en spécifications de bas-niveau. Par la suite, le générateur de code peut-être utilisé pour générer du code exécutable directement à partir des spécifications. Ceci permettant d'éviter une traduction des spécifications en code source pouvant mener à des erreurs. La Figure 3.3 montre le processus de développement proposé par SCADE (réalisé à partir du processus de développement de la DO-178B) dans lequel les étapes de spécifications, de conception et de codage (remplacé par la génération de code) sont maintenant faites avec SCADE.

3.3.2 Le processus de vérification avec SCADE

La vérification d'un design SCADE doit respecter les mêmes objectifs d'un projet suivant la DO-178B. Les objectifs principaux d'un tel projet ont déjà été décrits à la

section 3.2.2. Ce qui change, c'est la façon d'appliquer les méthodes de vérification. En effet avec SCADE, toutes les spécifications qui y sont décrites (haut-niveau et bas-niveau) peuvent être simulées et testées à partir de l'outil. Ceci permet de vérifier le fonctionnement de l'application avant l'implémentation. Ainsi, les analyses des spécifications, la vérification de leur fonctionnement et les tests logiciels peuvent être tous faits à partir de SCADE. De plus, à l'aide de l'outil Prover, il est possible de faire de l'analyse formelle des modèles (voir Chapitre 4 pour plus de détails à ce sujet).

En utilisant le générateur de code qualifiable (KCG), il est même possible d'éliminer les tests et la couverture structurelle du code. En effet en faisant l'usage de KCG, il est garanti que le code généré aura exactement le même comportement que les spécifications à partir duquel il a été créé. Lorsque la couverture structurelle a été vérifiée et les tests effectués pour les spécifications, de par la nature du compilateur KCG, le code généré hérite des mêmes caractéristiques de couverture et de tests. Le générateur a été qualifié aux normes de la DO-178B pour garantir son comportement et ainsi permettre d'éviter une série de tests sur le générateur. Si un autre générateur de code non-qualifiable avait été utilisé, le code aurait dû être vérifié et testé de la même manière que si il avait été fait « à la main ». En ce sens, le générateur de code aurait seulement été un outil plus efficace pour écrire du code.

3.3.3 Cycle en Y

Le cycle en V, tel que vu à la section 3.2.3, devient avec la méthodologie SCADE un cycle en Y. En effet, en utilisant le générateur de code, du code source en C est généré directement à partir des spécifications écrites en SCADE. De plus, KCG va permettre de supprimer la vérification, les tests unitaires et les tests de couverture du code source. La Figure 3.4 décrit ce nouveau cycle en Y. Tout comme le cycle en V, la partie gauche du cycle en Y montre les étapes d'écriture des spécifications

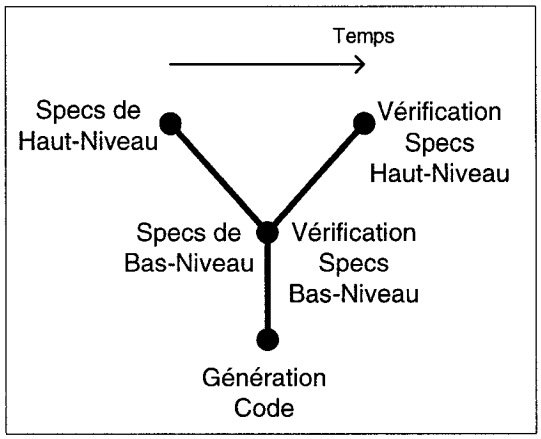


Figure 3.4 Cycle en Y

haut-niveau et des spécifications bas-niveau. L'écriture des spécifications bas-niveau se fait totalement en SCADE, tandis que l'écriture des spécifications haut-niveau peut se faire partiellement avec l'outil. À partir de ce point, l'étape suivante, la génération de code, devient instantanée. En effet, en utilisant le générateur KCG, le code est généré instantanément et la vérification du code n'est plus nécessaire. Donc la partie de droite du cycle étant la vérification, il ne reste qu'à vérifier les spécifications de bas-niveau et ensuite les spécifications de haut-niveau.

Le cycle en Y montre aussi une accélération du processus de développement d'applications critiques, comparativement au cycle en V. Ceci est principalement dû à l'utilisation du générateur de code KCG. Dans son projet du second système de contrôle de vol du A340/500, Airbus a pu générer 70% du code total de l'application. Ils ont mesuré une réduction de 50% dans les coûts de développement et une réduction du temps total du cycle de conception d'un facteur de 3 à 4 (comparé au codage manuel) ([43] et [15]). Eurocopter a pu générer 90 % du code des systèmes autopilotes des hélicoptères EC135 et EC155 et ils ont mesuré une réduction de temps de développement de l'ordre de 50% (comparé au codage manuel) [15].

3.3.4 Exemple du régulateur de vitesse

Cette application de régulation de vitesse est donnée dans le but de présenter un exemple d'application de la méthodologie SCADE. Cet exemple n'a pas pour objectif de spécifier ou de décrire le fonctionnement d'un vrai régulateur de vitesse tel qu'installé sur certaines automobiles, mais plutôt une simplification de ceux-ci. Il est directement inspiré de l'exemple du régulateur de vitesse conçu par Esterel-Technologies et fournit avec SCADE.

Tableau 3.1 Les spécifications de haut-niveau du module de gestion de la vitesse

Entrées:	**Sorties:**
- État du Régulateur (ON ou OFF) - Vitesse du Véhicule (entier) - Bouton de réglage de la vitesse (booléen) - Bouton d'accélération rapide (booléen) - Bouton de décélération rapide (booléen)	- Vitesse de croisière (entier)
Spécifications détaillées: a- Le module de gestion de la vitesse est actif si et seulement si le régulateur est activé (ON), sinon la vitesse de croisière ne change pas, c'est-à-dire garde la même valeur que la valeur précédente. b- Si le bouton de réglage de la vitesse est pressé[1], la vitesse de croisière est égale à la vitesse du véhicule, sinon : 1. Si le bouton d'accélération rapide est pressé[1], la vitesse de croisière est augmentée de *SpeedInc* km/h, seulement si la nouvelle valeur est plus petite que la vitesse maximale *SpeedMax* km/h. 2. Si le bouton de décélération rapide est pressé[1], la vitesse de croisière est diminuée de *SpeedInc* km/h, seulement si la nouvelle valeur est plus grande que la vitesse minimale *SpeedMin* km/h. 3. Si aucun bouton n'est pressé[1], la vitesse de croisière ne change pas, c'est-à-dire garde la même valeur que la valeur précédente	
Initialisation: - À l'initialisation du système, la vitesse de croisière est 0 km/h.	

[1] Du point de vue du type booléen, un bouton dit « pressé » est considéré comme étant à la valeur « vraie »

Le régulateur de vitesse est un système qui garde la vitesse d'une automobile

constante. Lorsque le régulateur est mis en marche, une vitesse de croisière doit être choisie avant que le système commence à maintenir la vitesse. La méthodologie SCADE sera appliquée sur un des modules du régulateur de vitesse, le module de gestion de la vitesse. En effet, à partir de la spécification haut-niveau du module écrite en texte, la spécification bas-niveau sera implantée à l'aide de SCADE et du code sera généré à l'aide de KCG.

Le module de gestion de la vitesse s'occupe de calculer la vitesse de croisière de la voiture (vitesse à laquelle elle sera maintenue) à partir de l'état du régulateur de vitesse (en marche ou éteint), de la vitesse actuelle et des commandes de contrôle qui sont activées. Le Tableau 3.1 présente les spécifications haut-niveau du module.

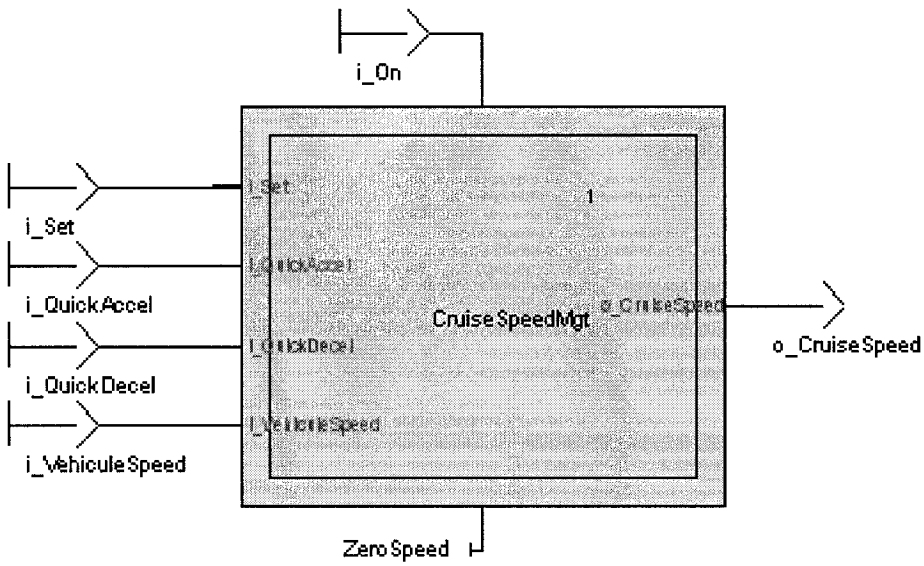


Figure 3.5 Interface du module de gestion de la vitesse

À partir des spécifications haut-niveau, le module a été implémenté avec SCADE, devenant par le fait même les spécifications bas-niveau. La Figure 3.5 montre l'interface du module. Les entrées du module ont été implémentées comme suit: état du régulateur (i_On), vitesse du véhicule (i_VehiculeSpeed), bouton du réglage de la vitesse (i_Set), bouton d'accélération rapide (i_QuickAccel) et bouton de décélération rapide (i_QuickDecel). La seule sortie du module étant la vitesse de

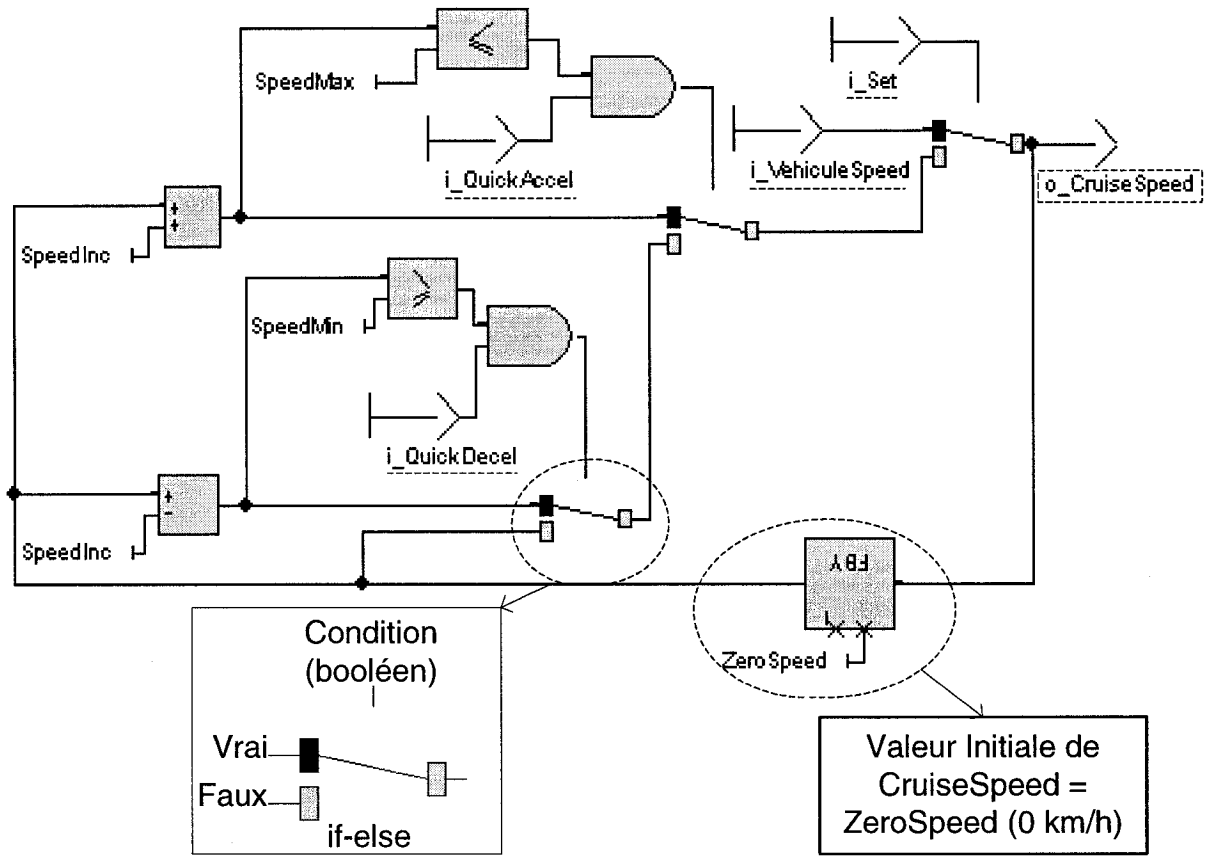


Figure 3.6 Implémentation du module de gestion de la vitesse

croisière (o_CruiseSpeed). La spécification a) au Tableau 3.1 a été implémentée à l'aide d'un condact (noeud conditionnel). En effet, le module de gestion de la vitesse ne sera exécuté que lorsque l'entrée i_on sera vraie (c'est-à-dire lorsque le régulateur sera en marche). La valeur d'initialisation du condact (c'est-à-dire la valeur initiale de la sortie o_CruiseSpeed) a été établie à *ZeroSpeed* (0 km/h)

La Figure 3.6 montre comment le reste des spécifications du module (b, b1, b2 et b3) ont été implémentées dans un diagramme bloc à l'aide d'une cascade de if-else. Chacune des conditions déterminant la valeur de la vitesse de croisière peut y être ainsi facilement exprimée. Pour améliorer la lisibilité du diagramme, les entrées du module ont été soulignés et la sortie encadrée. Le symbole des conditions if-else étant particulier à SCADE, un exemple de celui-ci a été ajouté au diagramme.

De plus, l'explication du symbole FYB (servant à faire l'initialisation de la sortie o_CruiseSpeed) a aussi été ajoutée. Le reste des symboles utilisés se décrivent facilement par eux-mêmes.

À partir du modèle bas-niveau du module dans SCADE, il devient possible de le simuler et de le vérifier à partir de l'outil. Une fois la simulation et la vérification terminées, du code C peut être généré à l'aide du générateur KCG. L'annexe A donne le code source généré du module de gestion de la vitesse (appelé Top_CruiseSpeedMgt). Comme expliqué à la section 3.3, ce code est exécutable et possède tous les avantages d'un code généré à partir d'un outil de génération qualifié selon la norme DO-178B.

3.3.5 Les avantages apportés par la méthodologie SCADE

L'avantage principal de la méthodologie SCADE est de couvrir complètement le cycle de développement à partir des spécifications jusqu'au code certifié. De plus, celle-ci essaie de prévenir plusieurs problèmes fréquemment rencontrés dans les autres méthodologies de développement. Par exemple, les descriptions multiples sont remplacées avec SCADE par un modèle commun qui peut être raffiné et partagé entre les membres d'un projet. Qui plus est, SCADE adresse trois types d'erreurs : le codage manuel est remplacé par le codage automatique, évitant ainsi les *erreurs de codage* et permettant un gain de productivité substantiel; les modèles SCADE peuvent être directement simulés et validés, la détection tardive d'*erreurs de design et de spécification* est beaucoup plus rare; le modèle aide à prévenir les *erreurs d'interprétation*, quand elles sont détectables, grâce à sa représentation formelle et intuitive [15].

Le modèle graphique de SCADE est considéré comme beaucoup plus intuitif qu'un modèle texte. Un ingénieur système utilisant SCADE n'a pas vraiment besoin de savoir coder en C ou en ADA. De plus, ce modèle graphique se rapproche beaucoup

des modèles utilisés par les ingénieurs en contrôle (SCADE étant souvent utilisé pour décrire des modules de contrôle) et peut être ainsi utilisé par ceux-ci. La section 3.4 rapporte une expérience d'utilisation faite avec SCADE pour vérifier la productivité de l'outil. Cette expérience donne une idée sur le caractère intuitif de l'outil.

3.3.6 Les inconvénients apportés par la méthodologie SCADE

L'utilisation de deux formalismes, LUSTRE pour le flux de données et les SYNCCHARTS pour la description du flux de commande, peut devenir un inconvénient pour la description de systèmes mixtes. Si le système à modéliser comprend une petite machine à états contrôlant de gros modules implémentant des lois de contrôle, ceci ne pose pas de problèmes. Par contre lorsque le système est vraiment mixte, l'utilisation des deux formalismes force une séparation forte entre le flux de commande et le flux de données qui n'est pas toujours évidente. [18] propose une solution à ce problème en ajoutant à LUSTRE des constructions impératives.

SCADE n'impose pas de techniques bien définies à suivre lors de la modélisation d'un système, donnant ainsi plus de liberté aux concepteurs. Par contre, ceux-ci peuvent alors modéliser leur système de façon désorganisée, réduisant ainsi la lisibilité et la réutilisation. [35] propose une méthodologie de design utilisant des « modes automata » qui permettent d'implémenter des modes d'exécutions. Ces modes d'exécutions permettent, par exemple, d'avoir plusieurs équations (modes) définissant la même sortie. Le bon mode étant choisi selon l'état du système.

3.4 Expérience d'utilisation de l'outil SCADE

Une expérience d'utilisation a été réalisée dans le cadre d'un laboratoire d'un cours gradué de systèmes embarqués à la Polytechnique de Montréal. Cette expérience avait

pour but de vérifier indirectement le caractère intuitif de l'outil. Plus précisément, une comparaison de la productivité de 32 étudiants travaillant en solitaire sur un design en appliquant une partie de la méthodologie SCADE et sur un design en C pour appliquer une méthodologie plus traditionnelle. En somme, l'expérience voulait vérifier si une des modélisations-méthodologies procurait un gain de productivité par rapport à l'autre. Les sujets ont été mis dans un contexte industriel où ils auraient à travailler sur un nouveau projet sans expérience au préalable sur le sujet du projet. Formellement, l'expérience se définit comme suit : « *L'analyse* ***de la modélisation avec SCADE et en C*** *dans le but* ***d'évaluer ces méthodes*** *en tenant compte de* ***l'efficacité et de la productivité*** *du point de vue* ***de l'expérimentateur*** *dans le contexte* ***d'étudiants du cours gradué de système embarqués travaillant sur les deux modélisations.***

Pour ce faire, les sujets ont été séparés au hasard en deux groupes (de 15 et de 17 étudiants) et ils ont effectué à l'intérieur de deux séances de laboratoire des manipulations sur un design C et SCADE d'un régulateur de vitesse. À la première séance, le premier groupe s'est concentré sur le design C et le deuxième sur le design SCADE. À la deuxième séance, ceux-ci ont inversé leur rôle. Cette expérience est en fait de type : « comparaison en paire » [54].

Chacun des sujets a reçu une formation de 3 heures sur SCADE avant de commencer leur laboratoire (en plus d'effectuer un Tutorial d'environ 45 minutes pour se familiariser à l'outil). À partir d'une spécification texte qui leur a été fournie (voir Annexe II Section II.1.1), les sujets ont effectué les manipulations suivantes :

1. Correction de deux bogues introduits dans le modèle
2. Ajout d'une fonctionnalité de gestion de la vitesse
3. Intégration du régulateur de vitesse à un système gestion des composants de la voiture fonctionnant à l'aide du système d'exploitation temps-réel μC/OS-II

Pour réaliser l'expérience, les sujets ont eu accès à un projet C ou un projet SCADE (selon la séance) sur lequel ils ont pu effectuer leurs manipulations. Une interface graphique leur a été fournie pour qu'ils puissent simuler plus facilement. De plus, un projet μC/OS-II implémentant le système de gestion des composants de la voiture leur a aussi été mis à disposition pour faire l'intégration finale.

Les sujets ont dû calculer le temps nécessaire pour effectuer leurs manipulations. Ce temps devenant la mesure de leur productivité. Pour des fins de comparaisons, il a été demandé aux sujets de ne calculer que le temps utile de travail (et non pas le temps pris pour comprendre la spécification ou de comprendre les énoncés). De plus, un questionnaire leur a été remis pour connaître leurs expériences passées et pour connaître leur appréciation générale des outils et des deux méthodologies utilisées.

Le but principal d'une expérience de ce type est de réfuter une hypothèse nulle à l'aide d'une analyse statistique des résultats. L'hypothèse nulle déterminée pour l'expérience est la suivante :

- *Il n'y a pas de différence de productivité entre la modélisation SCADE la modélisation C*

3.4.1 Résultats de l'expérience

L'annexe II présente les résultats de l'expérience. Il faut noter que dans le tableau final comparant les temps des deux séances pour chacun des étudiants (voir Tableau II.2), les temps de correction des bogues n'ont pas été ajoutés au temps final. Ceci vient du fait que la correction de ceux-ci était beaucoup plus rapide, lors de la deuxième séance, peu importe si c'était en C ou en SCADE (les bogues étant les mêmes pour les deux séances). Ceci dit, la correction des bogues a permis aux sujets de se mettre dans le contexte de l'application (c'était la première manipulation à effectuer) et d'avoir des résultats plus significatifs lors de la conception du module de gestion de la vitesse et de l'intégration.

Les résultats fournis par les sujets ont été analysés avec trois tests. Le test paramétrique: « test-t pour échantillons appariés » (voir section II.2.1.1) et les tests non paramétriques: « test Wilcoxon » (voir section II.2.1.2) et « test signé » (voir section II.2.1.3). Ces trois tests ont été choisis, car ceux-ci sont généralement utilisés dans l'analyse d'expériences comparatives, où chaque sujet compare deux manipulations faites avec deux méthodologies ou outils différents et, où une variable est analysée (ici la productivité en minutes). Ces tests essaient de trouver une tendance statistique qui permettrait de rejeter l'hypothèse nulle.

Après l'analyse des résultats des trois tests (voir sections correspondantes à chacun des test à l'annexe II), l'hypothèse nulle ne peut être statistiquement réfutée en utilisant un niveau de confiance (α) de 5%. Les résultats semblent nous indiquer que la productivité des étudiants n'est pas meilleure ou pire dans chacune des deux modélisations. A priori, il n'y aurait pas d'avantages du point de vue de la productivité à utiliser une modélisation plus que l'autre.

Les conclusions statistiques méritent d'être discutées. Après avoir analysé les questionnaires que les sujets ont remplis, on y trouve que la très grande majorité d'entre eux (90% soit 29/32) possédaient une bonne connaissance du langage C, contre aucun qui possédait une expérience préalable de SCADE. 10% (3/32) des sujets possédaient une bonne expérience avec la suite Mathlab Simulink et 50% (16/32) d'entre eux possédaient une expérience minime avec ce genre de logiciel (par exemple: un travail pratique à l'université). Quelconque avantage relié à la connaissance de Mathlab Simulink peut-être donc négligé dans les résultats.

En nombres absolus, 18 sujets sur les 32 ont eu de meilleurs temps avec SCADE qu'avec le langage C et un autre a eu des temps identiques. Ceci est impressionnant compte tenu de l'expérience des sujets avec le langage C. Ceux-ci ont pris au total 3684 minutes pour faire les manipulations SCADE et 4078 minutes pour celles en C.

Si on décompose ces nombres, on s'aperçoit que les sujets ont pris beaucoup plus de temps à modéliser la fonctionnalité CruiseSpeedMgt en SCADE (2511 minutes) qu'en C (1777 minutes). On peut penser ici que l'expérience des sujets avec le langage C a sûrement beaucoup influencé les temps.

À l'intégration, les sujets ont été beaucoup plus efficaces avec SCADE (1173 minutes) qu'en C (2301 minutes). Ceci est dû au fait qu'en SCADE, un adaptateur était généré pour faciliter l'intégration. Cet adaptateur permettait de créer la tâche implémentant le module du régulateur de vitesse ainsi qu'une fonction permettant de l'activer. En C, les sujets devaient eux-mêmes intégrer leur module dans uC, c'est-à-dire créer la tâche et créer un sémaphore permettant de réveiller celle-ci.

On pourrait s'imaginer qu'à compétence égale (C et SCADE), les résultats reliés à SCADE auraient été meilleurs, car le temps de modélisation de la fonctionnalité CruiseSpeedMgt aurait été plus court, et qu'on aurait probablement rejeter l'hypothèse nulle. Par contre, l'utilisation de l'adaptateur donne un avantage considérable à SCADE lors de l'intégration. Dans un projet où ce type d'adaptateur ne peut être utilisé, le gain de productivité relié à SCADE s'en trouve minimisé. Les sujets ont affirmé en grande majorité dans les questionnaires que la modélisation avec SCADE est beaucoup plus simple à comprendre et qu'il est plus facile d'obtenir une « vision complète » du modèle (compte tenu de l'aspect graphique de SCADE). Si on ne peut conclure catégoriquement à une augmentation de la productivité avec SCADE on peut affirmer avec un certain niveau de confiance que SCADE ne cause en contrepartie aucune perte de productivité.

Dans cette expérience, d'autres possibilités offertes par SCADE n'ont pas été comparées. Il serait bien de refaire une expérience de ce type et d'y ajouter des aspects de test et de vérification. Ainsi, on pourrait se faire une idée sur la productivité lors de l'utilisation complète de la méthodologie SCADE.

CHAPITRE 4

LA VÉRIFICATION AVEC SCADE

Ce chapitre présentera une méthodologie de vérification, dans un contexte de la norme DO-178B, qui peut être employée avec SCADE. Cette méthodologie permettra une utilisation des fonctionnalités avancées de SCADE soit le vérificateur de design (permettant l'analyse formelle) et l'outil de mesure de la couverture.

4.1 Fonctionnalités avancées de SCADE

SCADE propose des fonctionnalités avancées de vérification. La suite SCADE est l'une des premières à intégrer ce genre de possibilités, permettant ainsi de couvrir la majeure partie du flux de conception à l'intérieur d'un même outil. Dans les prochaines sections, l'analyse formelle et l'analyse de la couverture pouvant être réalisées avec SCADE seront décrites.

4.1.1 Analyse formelle

En informatique, l'analyse formelle se réalise à partir de la description mathématique d'un code source d'un programme. La sémantique du langage utilisé pour modéliser le programme doit être suffisamment rigoureuse pour permettre une analyse efficace. En effet, plus la description mathématique sera précise, plus évidente sera l'application de méthodes d'analyse formelle basées sur de la logique mathématique. Le langage SCADE/LUSTRE a, dès le départ, été conçu avec une sémantique formelle. Le modèle mathématique du flux temporel pour le flux de données dans LUSTRE en est un bon

exemple (voir section 1.2.1.2).

Les principales techniques d'analyse formelle, applicables sur des descriptions mathématiques de programmes écrits en langage tel que SCADE/LUSTRE, sont les suivantes :

- **La démonstration de théorèmes** : Technique consistant à réécrire des formules de la description mathématique à l'aide de règles d'inférence et de l'induction.
- **Diagrammes de décision binaire** : Technique consistant à représenter les formules booléennes à l'aide de diagrammes de décision binaire. Ceci permet d'obtenir une représentation plus compacte des formules qu'ils représentent.
- **SAT (Solutionneur de validabilité)** : Technique représentant un ensemble de méthodes visant à résoudre le problème de validité booléenne (c'est-à-dire trouver, à partir d'une fonction logique composée de clauses mathématiques et de variables, une façon d'assigner les valeurs « vrai » ou « faux » aux variables pour rendre la fonction vraie).

L'analyse formelle, réalisée dans une philosophie de vérification, permet de raisonner, à l'aide d'outils et de méthodes mathématiques, sur la façon dont les comportements ont été implémentés. Ces nouvelles possibilités de raisonnement rendent la vérification de systèmes beaucoup plus complète et performante.

4.1.1.1 Preuve vs test

Il a été vu à la section 3.2.2 que les tests logiciels sont utilisés durant le processus de vérification. Par contre, lorsqu'une méthodologie de vérification formelle est appliquée, les tests sont remplacés par un nouveau concept appelé « preuves ». En effet, faire de la vérification formelle implique d'effectuer des preuves de propriétés à

vérifier. Les propriétés sont tirées directement des spécifications et servent à décrire la réaction du système à des situations (par exemple, la spécification ne permet pas rentrer le train d'atterrissage quand l'avion est au sol). En fait, la preuve permet de prouver hors de tous doutes qu'une propriété est toujours vraie (ou toujours fausse).

La principale différence entre les tests et les preuves est le niveau de couverture qui peut être atteint. En effet, lorsqu'un programmeur écrit un test pour une propriété à vérifier, il est pratiquement toujours impossible de couvrir tous les cas possibles. Un test pour tous les cas possibles impliquerait d'écrire des tests pour chaque combinaison d'entrées d'un système. Généralement, le programmeur choisit une stratégie pour diminuer le nombre de tests. Ainsi, il se trouve à réduire le pourcentage de couverture. Le problème avec cette approche est qu'elle ne garantit pas la détection de tous les problèmes.

La preuve mathématique d'une propriété permet d'obtenir un taux de couverture de 100%. Normalement, une série de tests assure qu'un problème « auquel un designer de test a pensé » n'arrivera pas. Une courbe gaussienne peut être utilisée pour représenter les problèmes potentiels qui ont été vérifiés (le milieu de la courbe) et ceux qui ne l'ont pas été (les extrémités de la courbe). La preuve permet de s'assurer que tous les cas, incluant les extrémités de la courbe gaussienne, sont vérifiés. Ceci devient évidemment très avantageux, tout particulièrement pour les propriétés dites de sécurité (propriétés pour lesquelles la sécurité du système et/ou la sécurité des usagers du système sont en jeu).

4.1.1.2 Vérificateur de design

Le « Vérificateur de design » est l'outil permettant de faire la vérification formelle dans SCADE. Il est basé sur l'outil « Prover SL DE » de la firme Prover Technology [45] et il utilise la technique SAT pour effectuer les preuves de propriétés.

SCADE utilise des observateurs pour modéliser des propriétés. Un observateur est en fait un module qui observe le comportement d'un autre module et témoigne du comportement de celui-ci. L'observateur est écrit dans le même langage que le module en observation, permettant ainsi d'écrire des propriétés sans à avoir à maîtriser un autre langage. Dans SCADE, un observateur est un noeud avec des entrées et une sortie booléenne. L'observateur doit être conçu pour que cette sortie booléenne soit toujours vraie.

La Figure 4.1 donne un exemple où un observateur vérifie que le système de chauffage est toujours éteint lorsque la température du cockpit est plus grande que 20 °C. Ce noeud observateur sera alors branché au noeud « observé » et le rôle du vérificateur de design sera d'essayer de trouver un cas où la sortie de l'observateur devient fausse. Si celui-ci est incapable de le faire, cette propriété s'avérera alors toujours vraie et donc vérifiée. S'il existe un cas où la propriété devient fausse, le vérificateur de design donnera les valeurs d'entrées du noeud « observé » nécessaires pour obtenir ce cas.

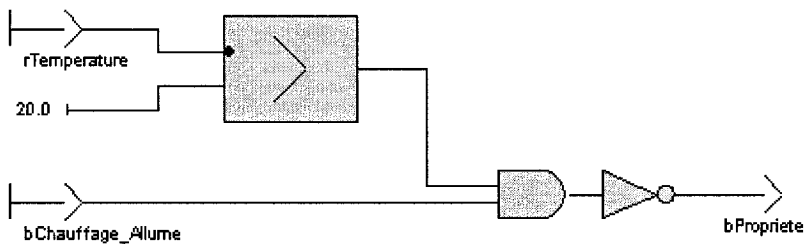


Figure 4.1 Exemple d'un observateur dans SCADE

Parfois, les valeurs d'entrées trouvées par le vérificateur de design, dans le cas où la propriété devient fausse, sont physiquement irréalistes. Il est alors possible d'utiliser les assertions de SCADE pour borner l'espace de valeurs réalistes qu'une entrée peut prendre (ex.: le capteur de température peut prendre des valeurs entre -25 °C et +50°C). Le vérificateur concentrera alors son analyse seulement sur les valeurs réalistes (voir Figure 4.2).

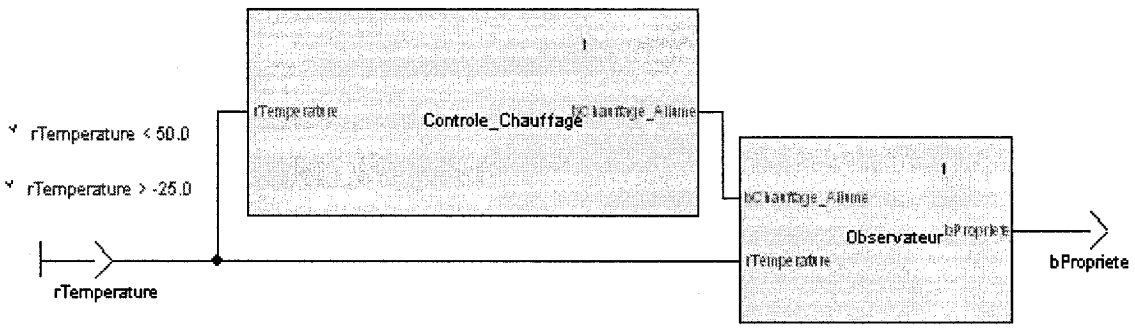


Figure 4.2 Exemple d'un noeud « observé » avec assertions

Le vérificateur de design a ses limites. Bien qu'il supporte complètement le langage SCADE, certaines propriétés ne peuvent pas être vérifiées. L'arithmétique non linéaire (fonctions mathématiques trigonométriques, exponentielles, racines carrées, etc.) n'est pas supportée. Dès qu'un noeud à être analysé comporte de l'arithmétique non linéaire, le vérificateur de design déclarera la propriété comme étant « indéterminée ». De la même façon, la multiplication est partiellement supportée (un opérande doit être une constante) pour les nombres réels et entiers. La division n'est pas supportée pour les nombres entiers et partiellement supportée (un opérande doit être une constante) pour les nombres réels. Pour essayer de contrer ces problèmes, le vérificateur de design permet le remplacement des fonctions non linéaires ou des noeuds problématiques par des opérateurs ou noeuds faisant des approximations des entrées et des sorties à l'aide d'assertions.

4.1.2 Analyse de la couverture

SCADE propose un outil de mesure de la couverture structurelle lors de l'application de cas de tests. Cet outil, le « Model Test Coverage » (MTC) permet de mesurer le pourcentage de couverture d'un cas de test. Ainsi, lorsqu'une série de tests est appliquée, la couverture structurelle peut être vérifiée. MTC permet de faire l'analyse de couverture de type « Couverture des Décisions » (DC) et de type « Couverture

modifiée de condition/décision » (MC/DC). Le glossaire de l'annexe III section III.2 donne les définitions des types de couverture structurelle selon la DO-178B.

Pour permettre une couverture à 100% de type DC, il faut, entre autres, que toutes les décisions menant à un branchement soient couvertes. Ici, deux cas de tests vérifiant les deux décisions possibles, le premier forçant CONDITION à VRAI et le deuxième à FAUX:

```
1. if CONDITION then
2. fait_quelque_chose
3. else
4. fait_quelque_chose_d'autre
5. end if
```

Une couverture à 100% de type MC/DC est atteinte lorsqu'il a été démontré, entre autres, que tous les résultats d'une décision ont été testés au moins une fois. Ensuite, il faut pouvoir démontrer que chaque condition dans une décision affecte de manière indépendante le résultat de cette décision. Par exemple, si la condition d'une décision est exprimée par (**A** OU (**B** ET **C**)), les cas de tests seront:

Tableau 4.1 Cas de tests MC/DC

Cas de test	**A**	**B**	**C**	**Résultat**
1	FAUX	FAUX	VRAI	FAUX
2	VRAI	FAUX	VRAI	VRAI
3	FAUX	VRAI	VRAI	VRAI
4	FAUX	VRAI	FAUX	FAUX

Ce jeu de tests permet d'affirmer que:

- Les cas de tests **1** et **2** démontrent que **A** affecte indépendamment le résultat de la décision;
- Les cas de tests **1** et **3** démontrent que **B** affecte indépendamment le résultat de la décision;
- Les cas de tests **3** et **4** démontrent que **C** affecte indépendamment le résultat de la décision;

L'atteinte d'une couverture structurelle de 100% est une condition de satisfiabilité des tests logiciels tels que décrits dans la DO-178B (voir [46] et section 3.2.2.1). La couverture de type instructions est obligatoire pour les logiciels de niveau C, la couverture de type DC pour le niveau B et la couverture MC/DC pour les logiciels les plus critiques, soit le niveau A (voir section 1.6.1 pour la description des niveaux).

Pour vérifier la couverture dans SCADE, il ne suffit que d'instrumenter le modèle (à l'aide de MTC) et d'écrire des scénarios de tests correspondants à des cas de tests. Lorsque MTC est démarré, celui-ci mesurera le pourcentage de couverture de chacun des noeuds lorsque les scénarios seront exécutés dans le simulateur.

4.2 Flux de conception utilisant des outils novateurs

Ces dernières années, de nouveaux outils comme le vérificateur de design et l'analyseur de couverture ont fait leur apparition sur le marché. Comme indiqué dans les sections précédentes, ces outils offrent de nouvelles possibilités très intéressantes. Un de leur plus grand avantage est leur intégration à une suite logicielle qui permet de couvrir une très grande partie du flux de conception. Cette caractéristique jumelée au formalisme du langage SCADE est la base de la valeur ajoutée par l'outil SCADE Suite.

Comme il a été vu au chapitre 3, SCADE peut être utilisé dans un flux de conception respectant la norme DO-178B. En revanche, les outils comme MTC et le vérificateur de design, ayant le potentiel d'être utilisés dans une méthodologie de vérification du flux de conception, ne sont pas encore largement exploités dans l'industrie. Une des raisons est sans doute l'absence, dans la littérature, d'une méthodologie utilisant ceux-ci dans un projet de conception avec SCADE. Les prochaines sections de ce chapitre essaieront de pallier ce problème en proposant un flux de conception respectant les objectifs de la DO-178B et faisant une utilisation de ces nouveaux outils.

4.2.1 Concepts

Pour la suite de la méthodologie, des précisions sur les spécifications doivent être apportées. Premièrement, comme il a été vu au chapitre 3, il est possible de décrire une certaine partie des spécifications haut-niveau (SHN) avec SCADE. La partie restante des SHN est écrite en texte. À partir de ces SHN textuelles, des spécifications bas-niveau (SBN) sont écrites avec SCADE. La DO-178B précise que : « si du code source doit être directement généré à partir des SHN, les SHN doivent être considérés comme étant des SBN et les directives des SBN doivent aussi s'appliquer ». En clair, les SHN écrites avec SCADE deviennent aussi des SBN (car du code source sera généré à partir d'elles) et la « vérification SBN » devra aussi s'appliquer à ces SHN.

4.2.2 Objectifs des activités de vérification

La dernière section de ce chapitre présentera un flux de conception qui mettra l'accent sur l'utilisation des outils de vérification de SCADE. La méthodologie de ce flux de conception respectera les objectifs des activités de vérification de la DO-178B. Ces objectifs ont été tabulés dans quatre tableaux à l'annexe A de la DO-178B [46] et ont été retranscrits à l'annexe III avec l'impact potentiel de l'utilisation de SCADE (ces impacts y sont aussi expliqués).

La première table, voir Tableau III.1 à l'annexe III, présente les treize objectifs devant être respectés lors des activités de vérification du processus de conception logiciel. Ces objectifs traitent, entre autres, de la qualité et de la vérifiabilité des SBN et de l'architecture logicielle. L'atteinte de la plupart des objectifs de cette table s'en trouve améliorée avec SCADE. Ceci étant dû a la notation SCADE, la sémantique formelle du langage, la structure du modèle SCADE et l'architecture relativement simple d'un tel modèle.

La deuxième table, voir Tableau III.2 à l'annexe III, présente les sept objectifs des activités de vérification concernant le code source. L'utilisation du générateur de code qualifiable (KCG) permet d'éliminer les six premiers. Seul le dernier objectif, sur le processus d'intégration du code source, ne peut être réalisé ou amélioré par SCADE.

La troisième table, voir Tableau III.3 à l'annexe III, présente les cinq objectifs des activités de vérification du code objet exécutable. La possibilité d'enregistrer des scénarios de simulation, pour vérifier le fonctionnement des modules SCADE avant la génération avec KCG, permet une réutilisation ultérieure de ceux-ci avec le code objet exécutable (sans à avoir le besoin de réécrire d'autres cas de tests). Cette réutilisation permet d'améliorer l'efficacité de l'exploitation des ressources accordées à la vérification.

La quatrième table, voir Tableau III.4 à l'annexe III, présente les huit objectifs des activités de vérification du processus de vérification. En particulier, la couverture structurelle des SHN, des SBN et du code. Il est possible de réutiliser les scénarios de tests employés dans la phase de design pour vérifier la couverture des SBN (ou des SHN si ceux-ci ont été implémentés dans SCADE) augmentant ainsi l'efficacité. L'utilisation de KCG permet d'éliminer tous les tests de couverture du code source, la couverture étant déjà vérifiée sur le design SCADE et l'apparition de « code mort » étant impossible avec KCG (voir annexe III section III.2 pour une définition du terme « code mort »).

La table 4.2 résume les objectifs devant être atteints par les activités de la méthodologie de vérification. Des objectifs initiaux, tous ceux ayant l'impact « éliminé » n'ont pas été traités. De plus, les objectifs du Tableau III.1 concernant l'architecture logicielle et le partitionnement logiciel ne seront pas considérés dans la méthodologie. Ceci étant par choix, la méthodologie se voulant une qui se concentre sur ce qui peut être accompli avec SCADE. Dans ce tableau, toutes les

Tableau 4.2 Objectifs pour la méthodologie de vérification

	Objectifs
1	Spécifications SCADE sont en conformité avec les Spécifications Système Textuelles
2	Spécifications SCADE sont précises et cohérentes
3	Spécifications SCADE sont compatibles avec la machine cible
4	Spécifications SCADE sont conformes aux standards
5	Spécifications SCADE sont traçables vers les Spécifications Système Textuelles
6	Les algorithmes sont précis
7	Le code objet exécutable est conforme aux Spécifications SCADE
8	Le code objet exécutable est robuste vis-à-vis des Spécifications SCADE
9	Les procédures de test sont correctes
10	Les résultats des tests sont corrects et les écarts expliqués
11	La couverture de test des Spécifications SCADE est assurée
12	La couverture structurelle (couplage contrôle/donnée) est assurée

spécifications SBN et SHN modélisées avec SCADE prennent le nom de « Spécification SCADE » (SS). Les SHN textuelles et les spécifications système (décrites à l'extérieur de SCADE) prennent le nom « Spécification Système Textuelle » (SST). Les objectifs concernant les SBN du Tableau III.1 devant aussi s'appliquer aux SHN modélisées avec SCADE, il était plus simple d'englober ces deux types de spécifications sous un même nom. Les objectifs concernant les SHN et les SBN du Tableau III.3 ont aussi été fusionnés, maintenant le code objet devant être conforme et robuste aux Spécifications faites avec SCADE. Dans le Tableau III.3, la vérification de la compatibilité avec l'ordinateur cible n'a pas été considérée, celle-ci étant réalisée à l'extérieur de SCADE.

4.2.3 Activités de vérification avec SCADE

Cette section décrit les activités de vérification qui doivent être accomplies avec SCADE et ses outils pour permettre d'atteindre chacun des objectifs du Tableau 4.2. La liste suivante énumère chacun des objectifs et la ou les activité(s) correspondante(s) [15].

[1] - SS sont en conformité avec les SST

L'utilisation de la notation graphique de SCADE rend l'écriture de SS qui sont en conformité avec les SST beaucoup plus facile grâce à sa nature fonctionnelle avec ses diagrammes blocs et ses machines à états. L'utilisation de cas de test avec le simulateur et la vérification de propriétés avec le vérificateur de design permet de vérifier la conformité des SS aux SST.

[2] - SS sont précises et cohérentes

Le langage SCADE a une sémantique formelle et utilise un typage fort, ceci a pour effet de rendre les modèles écrits en SCADE non ambigus et précis. Les erreurs d'interprétations sont aussi très réduites, car le modèle est utilisé par tous les participants du projet et celui-ci a la même signification pour chacun d'eux. L'utilisation de l'analyseur de design, vérifiant la syntaxe et la sémantique, permet de vérifier que le modèle reste conforme à la philosophie du langage.

[3] - SS sont compatibles avec la machine cible

Le générateur de code de SCADE permet de produire du code ANSI C (ou ADA) ayant un temps d'exécution constant et borné. Si ce temps est trop long, le modèle peut-être raffiné pour permettre des accélérations. De plus, l'utilisation du code ANSI C permet de s'assurer le code pourra être compilé par la grande majorité des compilateurs.

[4] - SS sont conformes aux standards

Le langage SCADE est lui-même un standard. La vérification de ses règles de sémantique et de syntaxe est effectuée par l'analyseur de design. Le générateur de code qualifiable garantit que ces règles sont respectées.

[5] - SS sont traçables vers les SST

L'utilisation d'un gestionnaire de spécifications (comme l'outil DOORS [51], pour lequel SCADE offre une passerelle) permet d'associer une SST à une ou

plusieurs SS. Ceci permet de tracer chacune des spécifications SCADE vers une spécification textuelle externe.

[6] - Les algorithmes sont précis

L'exactitude logique et temporelle est assurée « dès la construction » par l'utilisation du langage SCADE. Par contre, l'exactitude numérique doit être vérifiée à l'aide de tests, faits avec le simulateur, et la vérification formelle, faite avec le vérificateur de design.

[7] - Le code objet exécutable est conforme aux SS

Les scénarios de test et de simulation qui sont utilisés lors de la vérification des spécifications peuvent être réutilisés pour la vérification du code objet. Ceci permet de faire une vérification croisée du code objet avec des tests basés sur les spécifications.

[8] - Le code objet exécutable est robuste vis-à-vis des SS

La robustesse est vérifiée lors de tests, de la simulation et de la vérification formelle (par exemple la division par zéro) sur les spécifications SCADE. Le code objet exécutable obtenu à l'aide du code source généré par le générateur de code qualifiable (et d'un compilateur vérifié pour utilisation dans des projets critiques) est donc robuste dès la construction.

[9] - Les procédures de test sont correctes

La vérification des scénarios de tests utilisés par le simulateur permet de s'assurer que les procédures de test sont correctes.

[10] - Les résultats des tests sont corrects et les écarts expliqués

La vérification des résultats obtenus en simulation à l'aide du simulateur permet de s'assurer que les procédures de test sont correctes. La trace de la simulation étant gardée (fichier de sauvegarde excel), les divergences peuvent plus facilement être expliquées.

[11] - La couverture de test des SS est assurée

L'utilisation de l'outil d'analyse de couverture (MTC) lors de la simulation de cas de tests et de scénarios de test permet de vérifier la couverture de chacune des spécifications SCADE.

[12] - La couverture structurelle (données et contrôle) est assurée

L'utilisation de l'outil d'analyse de couverture (MTC) lors de la simulation de cas de tests et de scénarios de test permet de vérifier la couverture structurelle (selon le type et le niveau d'assurance du logiciel) de chacun des noeuds modélisant les spécifications SCADE.

4.2.4 Méthodologie permettant d'atteindre les objectifs de vérification

Le flux de conception présenté dans cette section mettra l'accent sur une méthodologie permettant de réaliser des activités de vérification avec SCADE respectant les objectifs de la norme DO-178B.

Le flux de conception se divise en trois grandes étapes. Premièrement, la modélisation SCADE à partir des spécifications systèmes textuelles, où seront écrits les spécifications SCADE, les propriétés formelles et les scénarios de tests. Par la suite, la vérification des spécifications SCADE permettra de faire la vérification formelle à l'aide des propriétés formelles, de faire la simulation à l'aide des scénarios de test et de faire une revue des spécifications. Finalement, la génération de code, permettra de générer le code ANSI C qualifiable à être intégré sur la cible. La Figure 4.3 présente le flux de conception SCADE proposé détaillant chacune de ces étapes.

La suite de cette section explique en détail chacune des activités de chacune des étapes du flux de conception proposé à la Figure 4.3. Lorsqu'il sera mention d'un « module », ceci fera référence à la fois à une fonctionnalité écrite en SCADE, à une hiérarchie de modules ou au design global (composé de hiérarchie de modules)

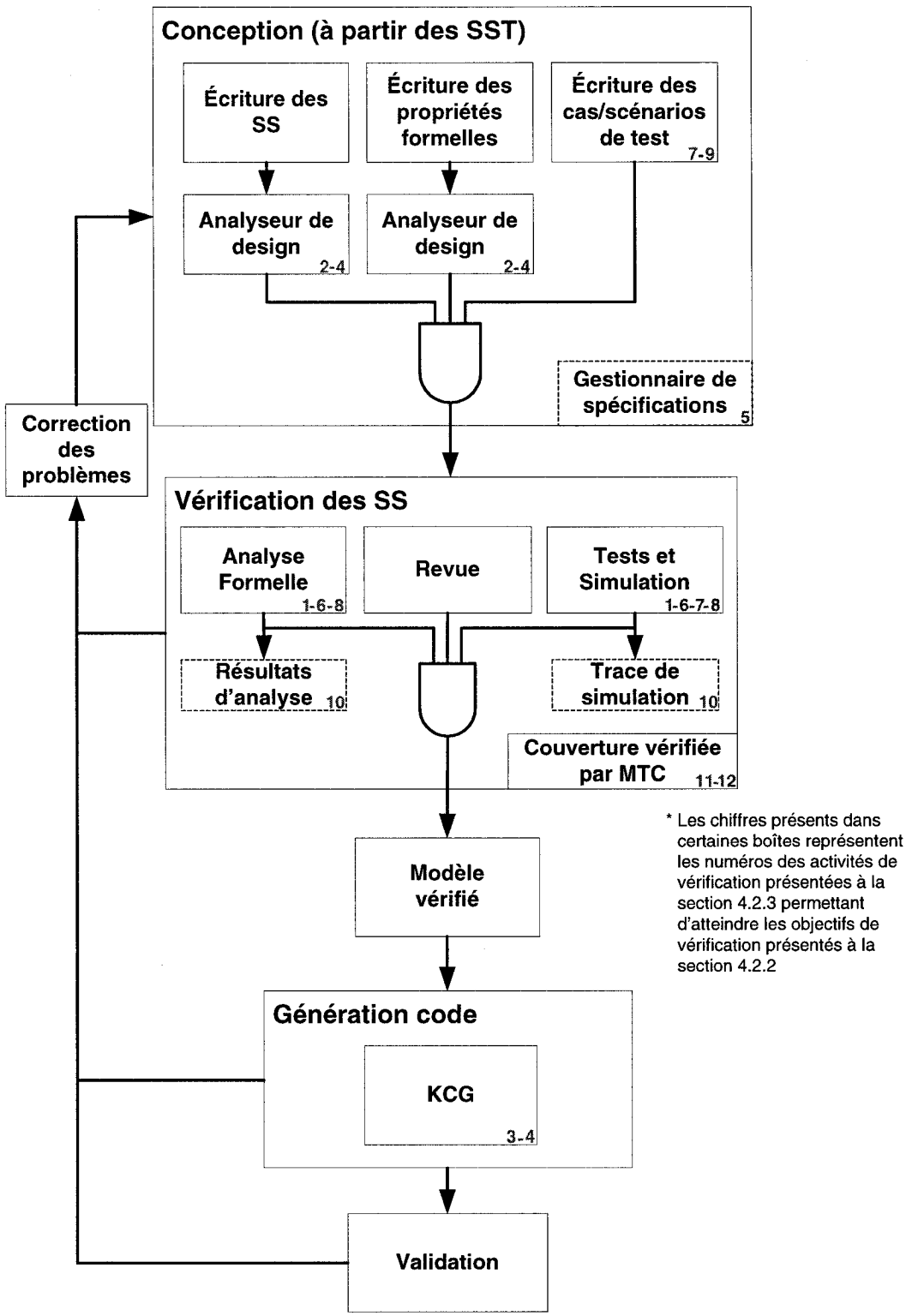


Figure 4.3 Le flux de conception SCADE proposé

Conception (à partir des SST)

La conception est l'étape où les spécifications système textuelles seront analysées et à partir desquelles seront écrites les spécifications SCADE, les propriétés formelles et les cas/scénarios de test. Les activités de conception se feront sur chacun des modules à tour de rôle. Par exemple, l'écriture des SS, l'écriture des propriétés formelles et l'écriture des cas/scénarios de tests seront effectuées pour le Module A, ensuite sur le Module B, etc.

Écriture des SS : À cette activité, chacune des SST sera précisée et formalisée en SS. Les SS doivent complètement décrire la fonctionnalité, pour que du code C pertinent puisse être généré à l'étape de génération de code.

Écriture des propriétés formelles : Ici, les propriétés formelles pouvant être tirées des SST et des SS seront écrites. En particulier, la vérification des plages de valeurs pouvant être envoyées au système. Les propriétés seront construites de façon à ce qu'elles vérifient qu'une caractéristique est toujours vraie. Durant la vérification formelle, le vérificateur de design essaiera de trouver un cas où la caractéristique devient fausse. Chacune des propriétés sera construite à l'aide d'un noeud observateur, branché au noeud sous vérification, et encapsulé dans un noeud test. Les entrées du noeud test seront bornées à l'aide d'assertion (pour s'assurer de faire l'analyse formelle sur des valeurs réalistes).

Analyseur de design : Durant cette activité, qui doit être réalisée après chaque écriture de SS et chaque écriture de propriétés formelles, la syntaxe et la sémantique seront analysées et les problèmes devront être corrigés. Cette activité permet de répondre aux objectifs de vérification 2 et 4.

Écriture des cas/scénarios de test : Ici, pour tous les cas où des propriétés formelles n'ont pas pu être écrites, des cas de tests ou des scénarios complets de tests doivent être rédigés. Ces tests sont tirés des SST et des SS. Une attention particulière doit être apportée au sujet de la

couverture structurelle. En effet, étant donné que celle-ci doit être vérifiée à la deuxième étape, les tests doivent tenter de couvrir complètement les SS. Les preuves n'assurant pas la couverture structurelle, une série de tests minimale permettant une couverture structurelle complète doit être ajoutée à chaque module où des propriétés formelles sont vérifiées. S'il est découvert à la vérification que la couverture des SS n'est pas assurée, d'autres tests devront être écrits. Cette activité permet de répondre aux objectifs de vérification 7 et 9.

Gestionnaire de spécifications : Pendant toute l'étape de la conception, un gestionnaire de spécification doit être utilisé pour permettre d'assurer la traçabilité entre les SST, les SS, les cas/scénarios de test et les propriétés formelles. Ceci permet de respecter le cinquième objectif de vérification.

Terminaison de l'étape : Une première itération de l'étape sera terminée quand toutes les SS, les propriétés formelles et les cas/scénarios de tests pour chacun des modules auront été écrits et analysés. Une fois ceci terminé, la prochaine étape, la vérification des SS, peut être amorcée. Si jamais des problèmes ou des inconsistances sont découverts à l'étape de vérification, il sera possible de revenir à l'étape de conception et de corriger ces problèmes.

Vérification des SS

La vérification est l'étape où on tente de détecter et de rapporter des erreurs qui ont pu s'introduire dans les SS durant la conception. Les activités de vérification se feront sur chacun des modules à tour de rôle. Par exemple, l'analyse formelle, la revue et les tests et simulation seront effectués sur le Module A, ensuite sur le Module B, etc.

Analyse formelle : Cette activité comprend l'utilisation du vérificateur de design pour vérifier les propriétés formelles décrites à l'étape de conception. Si un problème est détecté, l'étape de correction des problèmes devra être

effectuée. Cette activité adresse les objectifs de vérification 1, 6 et 8.

Revue : Les revues sont des analyses qualitatives des SS permettant de s'assurer que celles-ci sont conformes aux SST et que la documentation les décrivant est à jour. Les revues peuvent aussi servir à garantir que certaines règles établies pour assurer la robustesse ont été appliquées. Tous les points qui ne peuvent être vérifiés à l'aide de tests ou à l'aide de la vérification formelle doivent l'être durant les revues.

Tests et simulation : Les tests et la simulation comprennent l'exécution dans le simulateur de SCADE de cas de tests et de scénarios de tests décrits à l'étape de conception. Les scénarios de tests peuvent être par la suite utilisés par un simulateur externe à SCADE pour tester le code objet exécutable. Une première version de code pourra être générée si des règles de pire cas d'exécution doivent être examinées. Si un problème est détecté, l'étape de correction des problèmes devra être effectuée. Cette activité adresse les objectifs de vérification 1, 6, 7 et 8.

Résultats d'analyse : L'analyse des résultats de l'analyse formelle permet de s'assurer que tous les résultats de la vérification formelle sont corrects (c'est-à-dire qu'il ne reste plus de problèmes à corriger) et que les divergences sont expliquées. Dans le cas contraire, il faudra effectuer l'étape de correction et corriger ces problèmes. Ceci permet d'adresser le dixième objectif de la vérification.

Trace de simulation : L'analyse de la trace de simulation permet de s'assurer que tous les résultats des tests sont corrects (c'est-à-dire qu'il ne reste plus de problèmes à corriger) et que les divergences sont expliquées. Dans le cas contraire, il faudra effectuer l'étape de correction et corriger ces problèmes. Ceci permet d'adresser le dixième objectif de la vérification.

Couverture vérifiée par MTC : Le code des SS doit être instrumenté pour permettre l'analyse de couverture par MTC. En utilisant les cas/scénarios

de tests écrits à l'étape de conception, il faudra s'assurer que la couverture des SS et la couverture structurelle complète (selon le type et le niveau d'assurance du logiciel) de celles-ci sont atteintes. Si ce n'est pas le cas, l'étape de correction devra être effectuée et d'autres tests devront être écrits permettant de couvrir complètement les SS. Cette activité permet de respecter les objectifs 11 et 12 de la vérification.

Terminaison de l'étape : Une première itération de l'étape sera terminée quand la vérification de tous les SS aura été complètée. Si des problèmes ou des inconsistances sont découverts, il faudra retourner à l'étape de conception et les corriger. Par la suite, il faudra recommencer la vérification au complet pour vérifier les corrections et pour s'assurer qu'il n'y ait pas de régression. Une fois toute la vérification effectuée, tous les problèmes réglés et divergences expliquées, et la couverture complète atteinte, le modèle sera alors défini comme « vérifié ». À partir de ce point, la génération de code pourra être effectuée et le logiciel intégré à la cible.

Génération de code

La génération de code à l'aide de l'outil KCG permet de générer du code compatible avec la grande majorité des cibles et permet de s'assurer que les règles de sémantiques du langage ont été respectées. Si un problème survient, il faudra effectuer l'étape de correction des problèmes et repasser toutes les étapes avant la génération de code. La génération de code à l'aide de l'outil KCG permet respecter les objectifs 3 et 4 de vérification. Après cette étape, le code objet exécutable sera produit à partir d'un compilateur vérifié pour utilisation dans des projets critiques. Ainsi, d'autres activités de vérification pourront être effectuées pour répondre aux objectifs 7 et 8. Ces activités ne sont pas couvertes dans la présente méthodologie (car elles sont réalisées à l'extérieur de SCADE).

Validation

La vérification s'assure que le design est bien conçu et la validation s'assure que le bon design a été conçu (voir annexe III section III.2 pour une définition des termes validation et vérification selon la DO-178B). La validation finale s'effectue donc sur le vrai système complet. Par contre, grâce aux activités de revue des SS et de la simulation, un comportement inattendu peut déjà être détecté. En effet, il peut être découvert à ces étapes que le design construit n'est pas le bon. Comme toutes les étapes précédentes, si un problème est découvert, il faudra aller à l'étape de correction des problèmes et repasser toutes les étapes avant la validation.

Correction des problèmes

À chacune des étapes, si un problème est détecté, il faudra retourner à l'étape de conception (après une itération complète de l'étape en question) pour corriger le problème et recommencer tout le flux de conception. En effet, la DO-178B précise que la correction des erreurs est une activité du processus de développement (ici l'étape de la conception des SS).

La méthodologie présentée dans ce flux de conception permet de couvrir tous les objectifs de vérification présentés à la section 4.2.2. Le prochain chapitre présentera une application de la méthodologie dans un vrai projet industriel.

CHAPITRE 5

RÉSULTATS ET ANALYSE

Ce chapitre présente les résultats de la mise en application de la méthodologie de vérification présentée au chapitre 4. Un module de l'application CARP, utilisée en avionique, sera alors vérifié.

5.1 CARP

L'application CARP est utilisée pour faire du largage de chargement précis en plein vol. En effet, CARP calcule le point de largage d'un chargement dans l'air pour lui permettre d'atterrir dans une zone d'impact déterminée. L'acronyme CARP est défini par **C**omputed **A**ir **R**elease **P**oint. Cette application est utilisée par des avions (ex.: Hercules C-130) faisant du largage de chargement à basse altitude, le largage de chargement à haute altitude étant réalisé à l'aide de l'application HARP (**H**igh **A**ltitude **R**elease **P**oint). Le document AFI 11-231 [20] rédigé par la US Air Force, décrit les détails mathématiques et physiques de largage de CARP et HARP.

À l'aide de l'altitude de l'avion, de sa vitesse, de la pression atmosphérique, des coordonnées du terrain et des caractéristiques physiques du chargement et de son parachute, CARP peut calculer à partir de quelle coordonnée géographique (latitude et longitude) et à quelle vitesse le chargement doit être largué pour lui permettre d'atterrir dans la zone prédéfinie. Cette coordonnée géographique s'appelle le point de cheminement CARP (CRP). Lorsque ce point de cheminement est atteint, l'application se retrouve en mode Green Light où tout chargement, après largage, se retrouvera dans la zone spécifiée. Lorsque le point de cheminement XTE (Extended

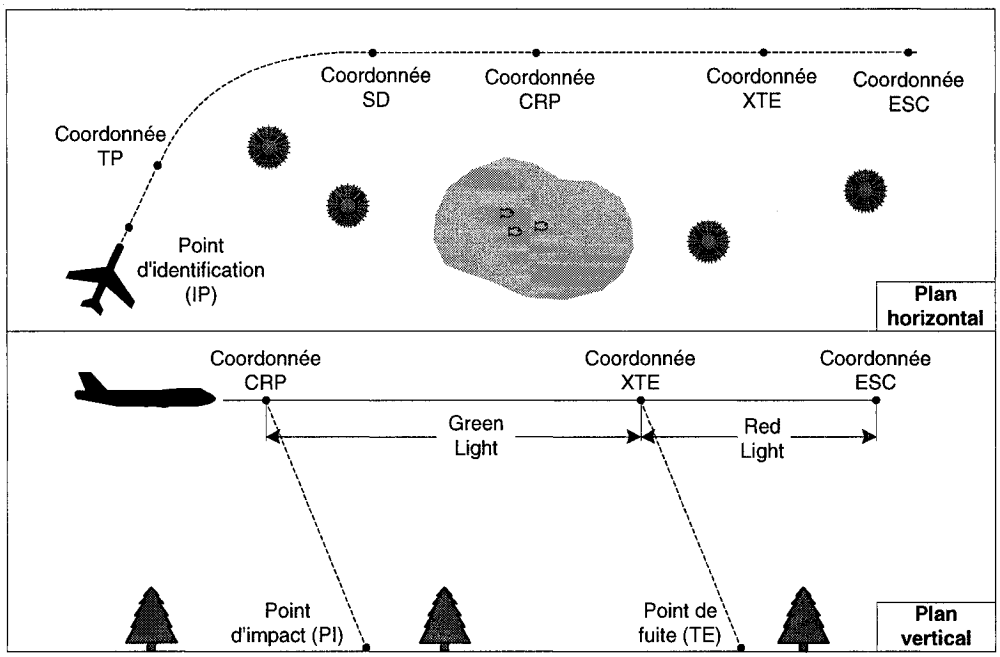


Figure 5.1 Agencement des points de cheminement de l'application CARP

Trailing Edge ou le bord de fuite étendu) est atteint, l'application se retrouve en mode Red Light et le largage de chargement doit être interrompu. La zone d'impact est définie par le point d'impact (PI), définissant le début de la zone, et par le point bord de fuite (TE), définissant la fin de la zone. La Figure 5.1 montre l'agencement de ces points en plus des points IP, définissant la coordonnée où l'avion doit s'identifier, les points de cheminements TP, indiquant le moment où l'avion doit tourner pour se positionner au-dessus de la zone de largage, SD, indiquant le moment où l'avion doit commencer à ralentir pour atteindre sa vitesse de largage et le point ESC, définissant la position où l'avion peut sortir de la trajectoire CARP et où la zone Red Light et l'application CARP se terminent.

5.1.1 Gestionnaire de messages

Le gestionnaire de messages est un sous-module de l'application CARP permettant de générer les messages informatifs reliés à l'application. Ces messages permettent

au pilote de savoir si des problèmes sont survenus durant le calcul de la trajectoire de largage et si l'état de l'avion (vitesse, altitude) correspond aux caractéristiques physiques du largage. De plus, des messages indiquant au pilote le temps avant d'atteindre le point de cheminement CRP, la zone Green Light et la zone Red Light sont aussi générés.

Ce module sera utilisé dans les prochaines sections comme exemple industriel d'application de la méthodologie de vérification. Les spécifications système textuelles complètes du gestionnaire de messages sont présentées à l'annexe IV section IV.2. Celles-ci sont directement inspirées des spécifications d'une application CARP conçue par une entreprise de développement avionique de Montréal.

5.2 Résultats

Dans les prochaines sections, le module gestionnaire de messages sera utilisé pour appliquer la méthodologie de vérification présentée au chapitre 4. À partir des spécifications système textuelles, le gestionnaire de messages a été implémenté à l'aide de SCADE. En suivant la terminologie présentée au dernier paragraphe de la section 4.2.2, les SST qui ont été modélisées dans SCADE, sont devenues des spécifications SCADE. Cette transcription correspond à l'activité « d'écriture des SS » de la méthodologie (et comprends celle de « l'analyseur de design » où la syntaxe et sémantique est vérifiée pour les SS). La Figure 5.2 présente la matrice de traçabilité entre les SST et les SS. L'utilisation des matrices de traçabilité remplace, d'une façon simpliste, l'utilisation d'un gestionnaire de spécification qui permet démontrer la traçabilité entre les différents éléments du système, et ainsi de répondre à l'objectif de vérification #5.

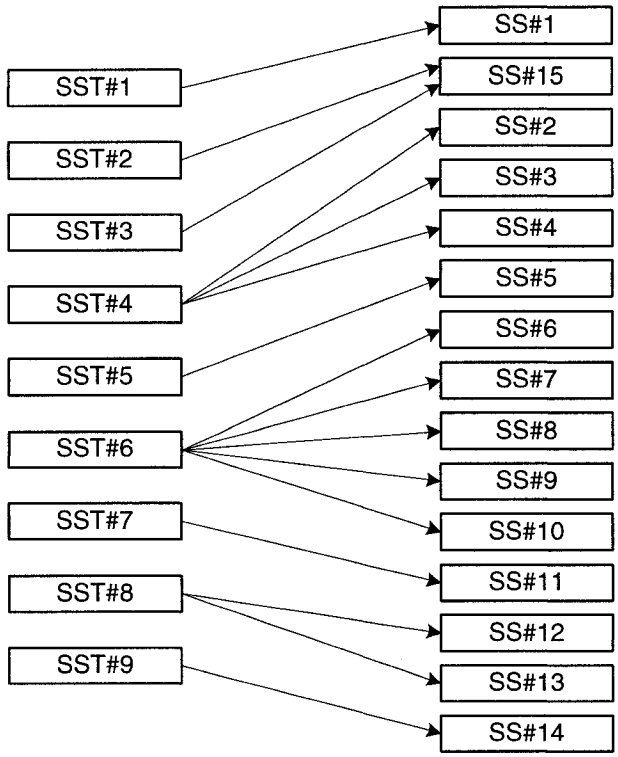


Figure 5.2 Matrice de traçabilité entre les SST et les SS

5.2.1 Description des manipulations

Les résultats expérimentaux ont été obtenus à partir deux différentes façons de mettre en oeuvre la méthodologie de vérification. Premièrement, celle-ci a été appliquée sans les activités d'écriture des propriétés et d'analyse formelle. En effet, ceci a permis d'obtenir un point de comparaison sous l'angle du nombre de cas de test nécessaires pour tester et vérifier chacune des SS. Cette façon de procéder correspond plus à l'industrie, où les tests sont abondamment utilisés. Cette omission des activités d'analyse formelle ne compromet aucunement l'atteinte des objectifs #1, #6 et #8 de vérification (voir section 4.2.2). Ceux-ci sont atteints à l'aide des activités de test.

Deuxièmement, la même méthodologie a été appliquée, mais cette fois-ci en y intégrant les activités d'écriture des propriétés et d'analyse formelle. Ceci a permis de démontrer jusqu'à quel point les preuves de propriétés formelles peuvent remplacer

les tests. Dans ce cas-ci, les métriques ont été le nombre de propriétés formelles, le nombre de cas de test remplacés par les preuves et le nombre de tests restant.

5.2.2 Manipulations sans l'analyse formelle

Dans cette première série de manipulations, la méthodologie de vérification a été complètement appliquée sans l'utilisation des propriétés et de l'analyse formelle. Pour ce faire, à partir des SST, une série de tests ont été écrits durant l'activité d'écriture des cas/scénarios de test. Ces tests ont permis d'adresser chacune des fonctionnalités décrites par les SST ainsi que les cas limites et les cas potentiellement problématiques. Chacun de ces tests a été tabulé à la section IV.3 de l'annexe IV.

Initialement, 151 cas de test avaient été écrits. Lors de la première itération de l'étape de vérification à l'activité de simulation, quelques problèmes de conception ont été trouvés dans le modèle, après l'analyse de la trace de simulation, obligeant ainsi à retourner à l'étape de conception pour les corriger. Lors de la deuxième itération, la correction des problèmes a été vérifiée. Par la suite, l'analyse de la couverture structurelle des cas de test avec l'outil MTC, selon le critère MC/DC (correspondant au niveau A de la DO-178B), n'a pu démontrer une couverture complète de 100% (les 151 cas de test ne couvraient environ que 92% des SS). Une troisième itération a été nécessaire pour faire l'ajout de 16 cas de test (SST#6f_T1ef (2 tests) et SST#6h_T4 (14 tests)). L'analyse de la couverture structurelle avec les **167 tests** a permis de démontrer une couverture complète de 100% avec le critère MC/DC. La revue des SS et des tests n'a pas permis de trouver d'autres problèmes et l'étape de vérification a pu être complétée. Ensuite, la génération de code, avec le générateur KCG, a pu être effectuée, répondant ainsi aux objectifs #3 et #4 de la vérification. Le Tableau 5.1 donne la liste des tests et le nombre de cas de test impliqués. La Figure 5.3 présente la matrice de traçabilité entre les SST, les SS et les tests.

Tableau 5.1 Liste des tests et du nombre de cas de test impliqués

ID des tests	Nombre de cas de test	ID des tests	Nombre de cas de test
SST#1a_T[1,2,3,4]	16	SST#5d_T1	3
SST#1b_T[1,2,3,4]	16	SST#6a_T1	4
SST#2_T1	6	SST#6b_T1	4
SST#3_T[1,2]	6	SST#6c_T1	4
SST#4a_T1	4	SST#6d_T1	6
SST#4b_T1	4	SST#6e_T1	6
SST#4c_T1	4	SST#6f_T1	8
SST#4d_T1	4	SST#6g_T1	6
SST#4e_T1	4	SST#6h_T[1,2,3,4]	32
SST#4f_T1	4	SST#7_T1	6
SST#5a_T1	2	SST#8_T1	5
SST#5b_T1	2	SST#8_T1	9
SST#5c_T1	2	***TOTAL***	***167***

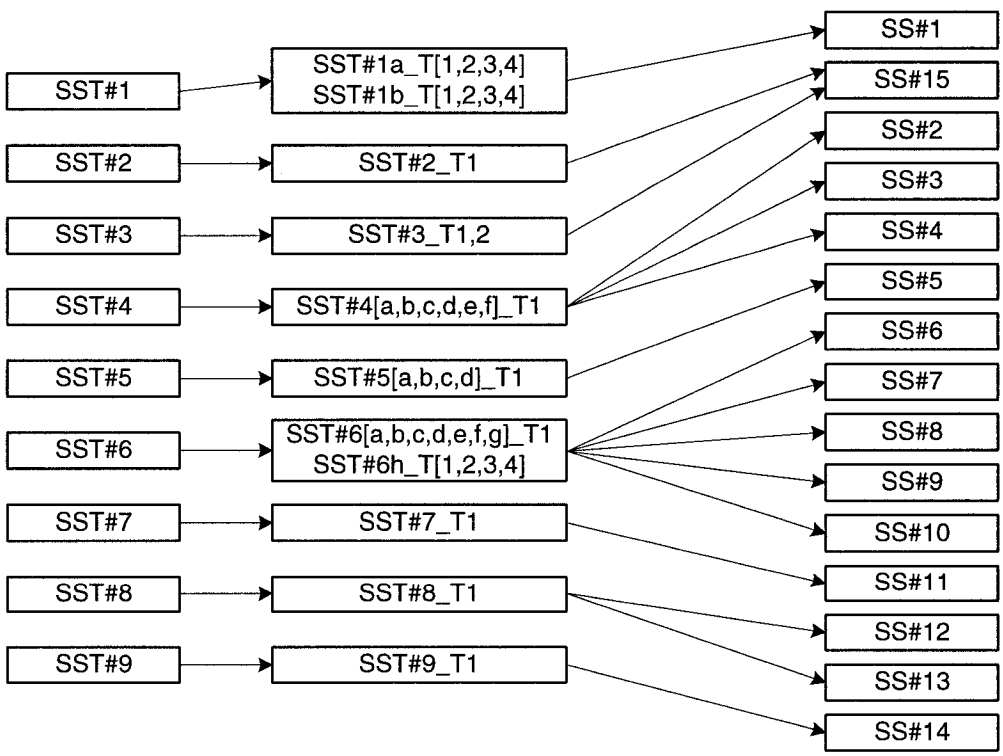


Figure 5.3 Matrice de traçabilité entre les SST, les tests et les SS

5.2.3 Manipulations avec l'analyse formelle

La méthodologie complète de vérification a encore une fois été appliquée pour la deuxième série de manipulation. Cette fois-ci, l'analyse formelle a été réintégrée à la méthodologie. À partir des SST, des propriétés formelles ont été écrites. Tout comme les tests, les propriétés ont essayé d'adresser les fonctionnalités décrites par les SST. Les propriétés ont été modélisées avant les tests pour permettre d'avoir le plus de propriétés possible vérifiant le plus de fonctionnalités possibles décrites dans les SST. Une propriété est toujours écrite à l'aide d'un noeud observateur dans SCADE. Celui-ci doit avoir une sortie qui doit toujours être VRAIE si la propriété est vérifiée.

À partir des SST, 35 propriétés formelles ont pu être écrites. Le gestionnaire de messages se prête bien à l'écriture de propriétés formelles, car sa fonctionnalité principale est de générer des messages lorsque certaines conditions sont remplies. Donc, plusieurs de ces propriétés s'assurent qu'une action sera toujours effectuée (émission d'un message) si les conditions menant à cette action sont remplies. D'autres vérifient l'inverse, où une action ne doit jamais être effectuée si les conditions menant à cette action ne sont pas remplies. Certaines d'entre elles vérifient que deux signaux ne peuvent pas être émis en même temps. Ce type de vérification et celui assurant l'absence sous toutes les conditions sont très difficiles à réaliser à l'aide de simples tests et souvent même impossibles, car ceci implique l'écriture de tests pour chaque combinaison d'entrées possible.

Une fois l'étape d'écriture de propriétés complétée, 54 cas de test ont été écrits pour finir la couverture des fonctionnalités décrites dans les SST. Lors de la première itération de vérification, l'analyse formelle a permis de repérer un problème avec la machine à états du noeud AT_CARP_State (SS#13) où le message AT_CARP pouvait être généré pendant un cycle quand la RED LIGHT était atteinte (ce qui est contraire à SST#8b). Ce problème est très intéressant du fait qu'il n'avait pas

Tableau 5.2 Liste des tests et du nombre de cas de test impliqués

ID des tests	ID de la propriété remplaçante	Nombre de cas de test	Nombre de tests ajoutés
SST#1a_T[1,2,3,4]	SST#1a_Pro[1,2]	0	0
SST#1b_T[1,2,3,4]	SST#1b_Pro[1,2]	0	0
SST#2_T1	SST#2_Pro1	0	0
SST#3_T1	SST#3_Pro1	0	1
SST#3_T2	-	3	0
SST#4a_T1	SST#4a_Pro1	0	3
SST#4b_T1	SST#4b_Pro1	0	0
SST#4c_T1	SST#4c_Pro1	0	2
SST#4d_T1	SST#4d_Pro1	0	0
SST#4e_T1	-	4	0
SST#4f_T1	SST#4f_Pro1	0	0
SST#4 (pas de test correspondant)	SST#4x_Pro[1,2,3,4,5]	0	0
SST#5a_T1	SST#5a_Pro1	0	2
SST#5b_T1	SST#5b_Pro1	0	2
SST#5c_T1	SST#5c_Pro1	0	2
SST#5d_T1	SST#5d_Pro1	0	2
SST#6[a,b,c,d,e,f]_T1	SST#6[a,b,c,d,e,f]_Pro[1,2]	0	0
SST#6g_T1	-	6	0
SST#6h_T[1,2,3,4]	-	32	0
SST#7_T1	SST#7_Pro1	0	1
SST#8_T1	SST#8_Pro[1,2]	0	3
SST#9_T1	-	9	0
TOTAL	**35**	**54**	**18**

été détecté lors de la première manipulation (avec la méthodologie sans l'analyse formelle). Après la deuxième itération où le problème avec le noeud SS#13 a été réglé, l'analyse formelle et les tests ont permis de conclure qu'il n'y avait plus de problèmes détectables. Par contre, l'analyse de la couverture structurelle des 54 cas de test restant, selon le critère MC/DC avec MTC, a révélé un taux de couverture à 95%. L'atteinte d'un taux de 100% étant un des objectifs de la vérification, il a fallu retourner à l'étape de conception pour y écrire les tests manquants et ainsi compléter la couverture. Et ce, même si certains tests et certaines propriétés

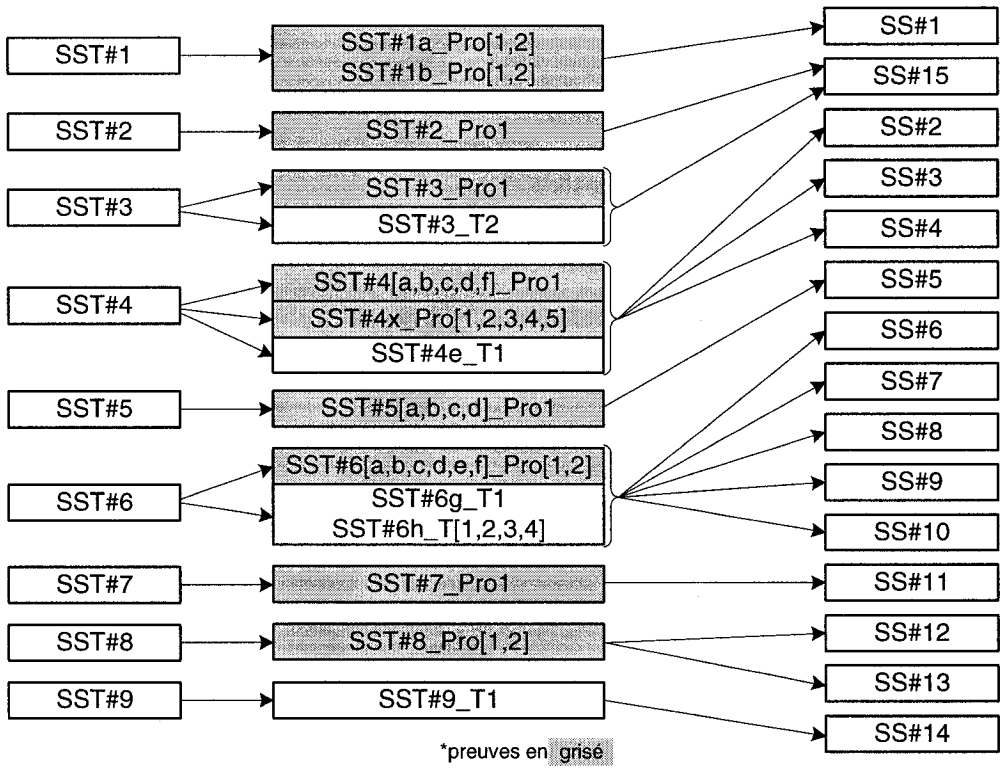


Figure 5.4 Matrice de traçabilité entre les SST, les tests, les propriétés et les SS

formelles s'entrecoupent (c.-à-dire vérifie la même fonctionnalité). Il faut savoir que la vérification d'une propriété formelle à l'aide du vérificateur de design, ne permet pas d'augmenter le taux de couverture structurelle, d'où le besoin d'entrecouper les propriétés et certains tests. La troisième itération de vérification a permis de montrer qu'avec 18 cas de test de plus, le taux de couverture de 100% pouvait être atteint. À partir de ce point, la revue des SS, des tests et des propriétés formelles n'a pas permis de trouver aucun autre problème et l'étape de vérification a pu être complétée. Par la suite, le code a pu être généré à l'aide de KCG.

Le Tableau 5.2 donne la liste des tests remplacés par des propriétés formelles, ainsi que les tests ajoutés pour permettre une couverture complète. Les cinq preuves SST#4x_Pro[1,2,3,4,5] ne correspondent à aucun test. Ces propriétés vérifient que deux signaux ne sont jamais émis en même temps, ce qui était impossible à vérifier

avec les cas de test. La Figure 5.4 présente la matrice de traçabilité entre les SST, les SS, les tests et les propriétés formelles.

5.3 Analyse

Le Tableau 5.3 présente une vue récapitulative des résultats obtenus lors des manipulations des sections précédentes avec et sans l'analyse formelle. Au premier coup d'oeil, la constatation la plus évidente est que l'utilisation des preuves formelles a permis de réduire le nombre de tests de plus de la moitié. Plus particulièrement, 30 propriétés formelles ont permis de remplacer 113 cas de test et 5 propriétés formelles ont permis de faire une vérification qui était impossible à faire à l'aide de simples tests.

Tableau 5.3 Tableau récapitulatif des résultats

	Nombre de cas de test	**Nombre de tests ajoutés (MC/DC)**	**TOTAL**	**Nombre de propriétés formelles**
Méthodologie sans analyse formelle	151	16	167	-
Méthodologie avec analyse formelle	54	18	72	35

Ces résultats indiquent que l'usage des preuves formelles peut être très intéressant. En effet, même si la métrique de temps de conception des tests versus le temps de conception des propriétés formelles n'est pas présentée ici, ils ont été en pratique quasi identiques. Certes, souvent il faut réfléchir un peu plus pour être capable d'écrire une propriété avec une sortie qui est toujours VRAIE et qui vérifie une fonctionnalité des SST, mais cet effort de réflexion se compare à l'effort nécessaire pour trouver une série de valeurs d'entrée pour permettre le test du système dans un état particulier. De plus, le temps d'analyse des 35 propriétés par le vérificateur de design ne prend

que quelques secondes. Par contre, dans des cas plus complexes, ce temps peut être beaucoup plus grand.

Le principal avantage des preuves est de prouver mathématiquement hors de tous doutes qu'une situation ne se produira pas. Souvent, cette assurance est quasi impossible à obtenir avec les cas de test. Par contre, les preuves ne peuvent pas être utilisées dans le domaine non linéaire et sont partiellement supportées pour certaines opérations mathématiques dans le domaine linéaire [11]. Le gestionnaire de messages se prête très bien aux preuves, étant donné son très grand nombre d'équations booléennes (qui sont complètement supportées par le vérificateur de design). Le noeud SS#1, qui implémente la SST#1, est complètement vérifié par les preuves formelles. Ce noeud, qui n'est composé que d'équations booléennes, s'occupe de générer deux messages lorsque certaines conditions sont présentes. En revanche, aucune propriété n'a pu être écrite pour le noeud SS#14, qui implémente la SST#9. Ce noeud, qui calcule la distance entre deux coordonnées, est composé d'équations non linéaires sur des nombres réels. SCADE permet de remplacer des équations non linéaires par des estimations linéaires qui peuvent être prouvées, mais dans le cas du noeud SS#14, ceci n'aurait pas permis l'écriture de propriétés. Donc, le vérificateur de design a une très grande valeur ajoutée pour les projets où beaucoup de logique booléenne est impliquée.

À certaines occasions, la façon dont la SST a été modélisée a empêché l'utilisation des preuves. Par exemple, les SST#4e qui a été implémentée à l'aide d'une machine à états, n'a jamais pu être exprimée en propriété du genre: « le message est toujours généré lorsque les conditions sont remplies ». Pourtant, les conditions de génération de tous les autres messages de ce noeud ont pu être exprimées en propriété et prouvées. Dans le même ordre d'idée, la preuve de l'impossibilité de la présence simultanée du message Red Light avec tout autre message de SST#4 n'a jamais pu être faite. L'analyse de la propriété par le vérificateur de design n'a jamais pu converger vers

une réponse, bien que cette propriété a pu être prouvée avec tous les autres messages (SST#4x_Pro[1,2,3,4,5]).

Cette analyse essaie de démontrer que dans tous les genres de projet, les preuves devraient être utilisées au maximum à cause des avantages qu'elles procurent. La conception des propriétés ne prenant pas nécessairement plus de temps que les tests, l'argument du temps ne devrait pas être utilisé. Par contre, avec la condition de couverture complète exigée par la DO-178B, les preuves ne pourront jamais éliminer complètement les tests. Il y sera toujours nécessaire d'effectuer une certaine quantité minimale de tests pour permettre d'atteindre la couverture complète (selon le critère désiré). C'est la raison pour laquelle il est très important de commencer à écrire les propriétés formelles, avant d'écrire les tests. Ceci permet l'usage des propriétés au maximum et l'usage des tests au minimum.

Les deux séries de manipulations ont aussi permis de démontrer que l'usage de la méthodologie de vérification proposée permet d'atteindre les objectifs de vérification recommandés par la DO-178B. Cette méthodologie se veut une méthodologie simple, facile d'utilisation et formative pour les gens qui débutent dans le monde de la conception et de la vérification encadrée par la DO-178B.

CONCLUSION ET TRAVAUX FUTURS

Après avoir fait une analyse des possibilités offertes par SCADE, de sa méthodologie de conception et de la section de la DO-178B sur la vérification, une série d'objectifs et d'activités respectant la DO-178B [46] et inspirée des possibilités offertes par SCADE a pu être déterminée. Ceux-ci ont servi de base à la méthodologie de vérification proposée au chapitre 4. Cette méthodologie a pour principal avantage de permettre l'utilisation des fonctionnalités avancées de vérification proposée par l'outil SCADE Suite. Celle-ci, répondant à l'objectif principal de ce projet, a été éprouvée à l'aide d'une fonctionnalité d'un exemple industriel et comparée à une méthodologie plus traditionnelle, n'utilisant que des tests pour faire la vérification.

L'utilisation de cette méthodologie peut, en plus de permettre une utilisation des fonctionnalités avancées de vérification offertes par SCADE, aider des gens qui souhaitent avoir une introduction à la vérification dans un contexte de DO-178B. En effet, la méthodologie synthétise et résume bien les idées générales et les objectifs tirés de la DO-178B. Celle-ci pourrait très bien être introduite dans un cours de modélisation de systèmes embarquées et appliquée à un travail pratique réalisé par des étudiants.

L'expérience d'utilisation a permis de constater que l'outil SCADE Suite ne semble pas influencer négativement la productivité. Au contraire, même sans expérience préalable, les participants à l'expérience ont obtenu une productivité comparable à celle d'un projet plus conventionnel en C. Sous toutes réserves, il ne serait pas faux de croire que si les participants avaient eu une compétance comparable avec l'outil SCADE et le langage C, la productivité de SCADE aurait été meilleure. Dans ce cas, il aurait probablement été possible d'affirmer, statistiquement, que l'outil SCADE procure un gain de productivité.

Rappel des contributions

La principale contribution de ce projet est de proposer une méthodologie de vérification qui fait usage des fonctionnalités avancées de l'outil SCADE Suite. La méthodologie a été conçue à partir d'objectifs de vérification tirés de la DO-178B et d'une série d'activité de vérification proposée par Esterel-Technologies. Actuellement, SCADE Suite est utilisé dans l'industrie, mais ses fonctionnalités avancées sont souvent ignorées, par manque d'exemple. L'application de la méthodologie sur le module du gestionnaire de messages présente, à la fois, un excellent exemple d'utilisation des fonctionnalités avancées de SCADE et une démonstration de l'efficacité de la méthodologie.

Également, l'expérience d'utilisation est aussi une contribution de ce projet. En effet, jusqu'à maintenant, aucune expérience de la sorte n'avait été menée avec l'outil SCADE Suite. Celle-ci pourra servir à rassurer des gestionnaires de projet qui hésiteraient à se lancer dans l'aventure SCADE.

Impressions personnelles sur SCADE Suite

En somme, travailler avec SCADE Suite fut une expérience très enrichissante. Les possibilités qu'offre l'outil sont très intéressantes. Les langages utilisés par SCADE (LUSTRE/SCADE et SYNCHARTS/SSM) offrent un formalisme de modélisation puissant, tout en limitant le concepteur à des constructions ne compromettant pas le déterminisme. L'utilisation du générateur de code qualifiable est aussi très profitable. Celui-ci permet aux concepteurs de se concentrer sur le design sans qu'ils aient à se soucier des détails d'implémentation. De plus, dans un contexte de DO-178B, il permet de réduire fortement le temps consacré à la conception et à la vérification. Dans le même ordre d'idées, le vérificateur de design propose une façon très intuitive de faire de la vérification formelle. Contrairement à d'autres types de vérification

formelle, le vérificateur de design n'impose pas l'apprentissage d'un autre langage ou d'un autre formalisme. En effet, les propriétés à vérifier sont modélisées avec des noeuds, dans le même style de conception que lorsque des fonctionnalités sont implémentées.

En revanche, l'adoption de la conception à la SCADE peut être assez laborieuse dans des projets où des programmeurs chevronnés y travaillent depuis longtemps. Le changement de philosophie peut être difficile à accepter. De plus, SCADE ne pouvant être utilisé pour tous les types d'applications (celui-ci étant beaucoup plus spécialisé dans les applications de contrôle), ceci peut impliquer l'utilisation de plusieurs conventions pour la conception d'un même projet. Par exemple, un projet en C pour les interfaces graphiques et pour le système d'exploitation temps réel et un projet en SCADE pour tout ce qui touche le contrôle. Bien sûr, SCADE permet la génération de fichiers adaptateurs pour faciliter l'intégration à certains systèmes d'exploitation temps réel et la possibilité de se brancher à une interface externe graphique pour faire de la simulation, mais ceci implique quand même l'obligation de maintenir plusieurs projets en parallèle.

Travaux futurs

Il serait intéressant d'utiliser la méthodologie de vérification dans un projet plus imposant. Ceci permettrait de vérifier si celle-ci tient la route pour un projet complet d'une taille normale en industrie. De plus, il n'est pas rare qu'en industrie, plusieurs personnes soient impliquées dans l'écriture des tests, dans l'analyse des résultats et dans les revues. Donc, l'utilisation de la méthodologie de vérification dans ce type de situation pourrait démontrer que celle-ci s'applique bien à ce cas.

Une méthodologie d'analyse de modèle et de conception de propriétés formelles permettrait de maximiser l'utilisation des preuves formelles. Cette méthodologie

plus technique pourrait proposer des constructions d'observateurs dans SCADE implémentant des types propriétés régulièrement vérifiés. En plus de s'approcher du nombre de tests minimal qui doit être produit pour obtenir une couverture structurelle complète, celle-ci permettrait de profiter au maximum des avantages que procurent les preuves formelles.

Pour faire suite à ce projet, une expérience d'utilisation réalisant la méthodologie de vérification permettrait de comparer le nombre de problèmes détectés par celle-ci à une autre méthodologie plus conventionnelle. L'autre méthodologie comparée n'aurait pas besoin d'être basée sur SCADE, mais bien sur n'importe lequel autre outil de vérification disponible. Ainsi, les forces et les faiblesses de SCADE en vérification pourraient être bien identifiées. L'expérience pourrait aussi concentrer son analyse sur le caractère intuitif des outils de vérification fournie par SCADE.

Dans un autre ordre d'idées, Esterel-Technologies [25] a annoncé de nouvelles possibilités pour SCADE dans les prochaines versions. En effet, en plus de générer du code logiciel, la génération de code matériel synthétisable (VHDL, SystemC) sera possible. Pour ce faire, le formalisme utilisé dans l'outil Esterel Studio sera complètement intégré à l'outil SCADE Suite. Donc, à partir d'un même modèle écrit en SCADE, du code logiciel ou matériel (au choix) pourra être généré. Il serait très intéressant de faire l'analyse de ces nouvelles possibilités dans un contexte conception codesign. Bien que le sujet n'ait pas été abordé dans ce mémoire, lorsque la performance du code logiciel généré devient problématique, peu d'options permettant d'accélérer le code sont offertes. La cogénération de ces modules en modules matériels permettrait de régler les problèmes de performances éventuels. De plus, ce nouveau générateur de code matériel devrait être qualifié. Donc, la proposition d'une méthodologie plus générale impliquant la conception codesign et la vérification dans un contexte de DO-178B et de DO-254 [47][1] pourrait s'avérer très intéressante.

[1]Norme encadrant la conception de modules matériels critiques en avionique

RÉFÉRENCES

[1] André, C. (2003). Modélisation de systèmes réactifs par une approche graphique synchrone: SyncCharts. (p. 15). École d'été Temps Réel, Toulouse (France).

[2] André, C. (2003). Semantics of S.S.M. (Safe State Machine). (p. 76). University of Nice-Sophia / CNRS.

[3] André, C., Boufaied, H. et Dissoubray, S. (1998). SyncCharts : un modèle graphique synchrone pour systèmes réactifs complexes. *RTS 1998.* (pp. 175–193). Paris: Actes de la conférence.

[4] Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Le Guernic, P. et de Simone, R., (2003). The synchronous languages 12 years later. *Proceedings of the IEEE, 91*(1), 64–83.

[5] Benveniste, A. et Le Guernic, P., (1990). Hybrid dynamical systems theory and the Signal language. *Automatic Control, IEEE Transactions on, 35*(5), 535–546.

[6] Berner, S., Joos, S., Glinz, M. et Arnold, M. (1998). A visualization concept for hierarchical object models. *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on.* (Vol. 00, pp. 225–228). IEEE Computer Society.

[7] Berry, G., (2003). The Effectiveness of Synchronous Languages for the Development of Safety-Critical Systems, 15. Consulté le 2006-07-26, tiré de http://www.esterel-technologies.com/files/Synchronous_Languages_Safety_Critical_Systems.zip.

[8] Bichler, L., Radermacher, A. et Schurr, A. (2002). Evaluating UML extensions for modeling real-time systems. *Object-Oriented Real-Time Dependable Systems,*

2002. (WORDS 2002). Proceedings of the Seventh International Workshop on. (pp. 271–278). San Diego, CA, USA.

[9] Binder, J., (2004). Safety-critical software for aerospace systems. *Aerospace America*, (August), 26–27.

[10] Booch, G., Jacobson, I. et Rumbaugh, J., (1997). *The Unified Modeling Language for Object-Oriented Development*, vol. v1.1. Rational Software Corp.

[11] Bouali, A. et Dion, B. (2004). Formal Verification for Model-Based Development. *SAE International.* Detroit, MI, USA.

[12] Boussinot, F. et de Simone, R., (1991). The ESTEREL language. *Proceedings of the IEEE*, *79*(9), 1293–1304.

[13] Boussinot, F. et de Simone, R., (1996). The SL synchronous language. *Software Engineering, IEEE Transactions on*, *22*(4), 256–266.

[14] Buck, D. et Rau, A., (2001). On modelling guidelines: Flowchart patterns for Stateflow. *Softwaretechnik-Trends*, *21*(2), 7–12.

[15] Camus, J.-L. et Dion, B., (2003). Efficient Development of Airbone Software with SCADE Suite, 49. Consulté le 2006-07-26, tiré de http://www.esterel-technologies.com/files/SCADE_DO-178B_2003.zip.

[16] Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S. et Niebert, P. (2003). From simulink to SCADElustre to TTA: a layered approach for distributed embedded applications. *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems.* (pp. 153–162). San Diego, California, USA.

[17] Clarke, E. M., J., Long, D. E. et McMillan, K. L., (1991). A language for compositional specification and verification of finite state hardware controllers. *Proceedings of the IEEE*, *79*(9), 1283–1292.

[18] Colaco, J. L., Pagano, B. et Pouzet, M. (2005). A Conservative Extension of Synchronous Data-flow with State Machines. *Proceedings of the 5th ACM international conference on Embedded software.* (pp. 173–182). Jersey City, NJ, USA.

[19] David, H. et Amnon, N., (1996). The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, *5*(4), 293–333.

[20] Departement of the Air Force, (2004). Computed Air Release Point Procedures (AFI11-231). United States of America, Air Force Publishing.

[21] Department of EECS, UC Berkeley. The Ptolemy Project. Consulté le 2006-09-20, tiré de http://ptolemy.eecs.berkeley.edu/.

[22] Espresso Team. Polychrony: a Toolset for Signal. Consulté le 2006-09-20, tiré de http://www.irisa.fr/espresso/Polychrony/.

[23] Esterel-Technologies. DO-178B Process Consulting. Consulté le 2006-09-19, tiré de http://www.esterel-technologies.com/services/process-consulting/do-178b-process-consulting.html.

[24] Esterel-Technologies. History. Consulté le 2006-09-20, tiré de http://www.esterel-technologies.com/company/history/.

[25] Esterel-Technologies. Voir site web. Consulté le 2006-09-20, tiré de http://www.esterel-technologies.com/.

[26] Federal Aviation Administration, (2006). Aircraft Certification. Consulté le 2006-09-19, tiré de http://www.faa.gov/licenses_certificates/aircraft_certification/.

[27] Ford, (1999). *Structured analysis and design using Matlab/Simulink/Stateflow - modeling style guidelines.* Technical report. Consulté le 2006-09-21, tiré de http://vehicle.me.berkeley.edu/mobies/papers/stylev242.pdf.

[28] Halbwachs, N. (1998). Synchronous programming of reactive systems, a tutorial and commented bibliography. *Synchronous Programming of Reactive System, A Tutorial and Commented Bibliography, CAV'98*. Vancouver B.C.

[29] Halbwachs, N. (2005). A synchronous language at work: the story of Lustre. *Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Proceedings.* (pp. 3–11). Verona, Italy.

[30] Halbwachs, N., Caspi, P., Raymond, P. et Pilaud, D., (1991). The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE, 79*(9), 1305–1320.

[31] Hamon, G. et Rushby, J. (2004). An operational semantics for stateflow. *FASE.* (pp. 229–243). Barcelona, Spain: Springer Verlag.

[32] Harel, D., (1987). Statecharts: A visual formalism for complex systems. *Sci. Comput. Program., 8*(3), 231–274.

[33] Harel, D. et Pnueli, A., (1985). On the development of reactive systems. *Logics and models of concurrent systems*, New York. Springer-verlag édition, pp. 477–498.

[34] IBM-Rational. Voir site web. Consulté le 2006-09-20, tiré de http://www-306.ibm.com/software/rational/.

[35] Labbani, O., Dekeyser, J.-L. et Boulet, P. (2005). Mode-Automata Based Methodology for Scade. *Hybrid Systems: Computation and Control (HSCC05).* (pp. 386–401). Zurich, Switzerland.

[36] LeGuernic, P., Gautier, T., Le Borgne, M. et Le Maire, C., (1991). Programming real-time applications with SIGNAL. *Proceedings of the IEEE, 79*(9), 1321–1336.

[37] Leveson, N. G., Heimdahl, M. P. E., Hildreth, H. et Reese, J. D., (1994). Requirements specification for process-control systems. *Software Engineering, IEEE Transactions on, 20*(9), 684–707.

[38] Mathworks, (2006). Stateflow and Stateflow Coder User's Guide v.6. Consulté le 2006-09-20, tiré de http://www.mathworks.com/access/helpdesk/help/pdf_doc/stateflow/sf_ug.pdf.

[39] Michael von der, B. (1994). A Comparison of Statecharts Variants. *Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems.* (pp. 128–148). Lübeck, Germany: Springer-Verlag.

[40] Micrium. Voir site web. Consulté le 2006-09-19, tiré de http://www.micrium.com/.

[41] OSEK-Siemens AG. Voir site web. Consulté le 2006-09-19, tiré de http://www.osek-vdx.org/.

[42] Per, B., Koen, C., Mary, S. et Satnam, S. (1998). Lava: hardware design in Haskell. *Proceedings of the third ACM SIGPLAN international conference on Functional programming.* (pp. 174–184). Baltimore, Maryland, United States: ACM Press.

[43] Pilarski, F. (1998). Cost effectiveness of formal methods in the development of avionics systems at Aerospatiale. *17th Digital Avionics Systems Conference.* (Vol. 1, pp. C14/1–C14/9). Seattle, WA.

[44] Potop-Butucaru, D., De Simone, R. et Talpin, J.-P., (2005). The synchronous hypothesis and synchronous languages. R. Zurawski, éditeur, *The Embedded Systems Handbook*, CRC Press.

[45] Prover Technology AB. Voir site web. Consulté le 2006-09-25, tiré de http://www.prover.com/.

[46] Radio Technical Commission for Aeronautics, (1992). Software Consideration in Airbone Systems and Equipment Certification. Federal Aviation Administration, USA, RTCA/DO-178B.

[47] Radio Technical Commission for Aeronautics, (2000). Design Assurance Guidance for Airborne Electronic Hardware. Federal Aviation Administration, USA, RTCA/DO-254.

[48] Seawright, A. et Brewer, F. (1992). Synthesis from production-based specifications. *DAC '92: Proceedings of the 29th ACM/IEEE conference on Design automation.* (pp. 194–199). Anaheim, California, United States: IEEE Computer Society Press.

[49] Siegel, S. et Jr., J. C., (1988). *Nonparametric Statistics for the behavioral sciences.* McGraw-Hill, New-York, seconde édition.

[50] Sundance Multiprocessor Technology Ltd. Voir site web. Consulté le 2006-09-20, tiré de http://www.sundance.com/.

[51] Telelogic AB. Voir site web. Consulté le 2006-09-19, tiré de http://www.telelogic.com/.

[52] Telelogic I-Logix Inc. Voir site web. Consulté le 2006-09-20, tiré de http://www.ilogix.com.

[53] The MathWorks, Inc. Voir site web. Consulté le 2006-09-20, tiré de http://www.mathworks.com.

[54] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Rgnell, B. et Wesslen, A., (2000). *Experimentation in software engineering: An Introduction.* Kluwer Academic Publishers, Norwell, Massachusetts.

[55] Wolf, E. S. (1995). Hierarchical models of synchronous circuits for formal verification and substitution. Thèse de doctorat, Stanford University, California.

[56] WoTUG. Occam Language. Consulté le 2006-09-19, tiré de http://www.wotug.org/occam/.

[57] Xia, Y. et Glinz, M. (2004). Extending a graphic modeling language to support partial and evolutionary specification. *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*. (Vol. 00, pp. 18–27). Busan, Korea.

ANNEXE I

CODE SOURCE

I.1 Module de gestion de la vitesse

Cette section présente le code source du module de gestion de la vitesse présenté à la section 3.3.4. Ce code source a été généré directement à partir du diagramme bloc du module CruiseSpeedMgt de l'exemple du régulateur de vitesse. Le générateur utilisé est le KCG version 4.2. La version de SCADE utilisée est SCADE Version 5.1 (build i13). La configuration par défaut « Qualifiable42 » a été utilisée pour faire la génération. Ici, seuls les fichiers « Top_CruiseSpeedMgt.c » et « CruiseSpeedMgt.c » sont présentés. Les autres fichiers étant moins intéressant.

« Top_CruiseSpeedMgt.c »

```
/*$************* SCADE_KCG KCG Version 4.2.0 (build i20) **************
** Command :
** l2C          CruiseControl.lus -node Top_CruiseSpeedMgt
**      -noexp @ALL@
**      -noassert
**      -const
**      -bitwise
**      -split
**      -no_copy_mem
** date of generation (MM/DD/YYYY): 08/01/2006 14:19:15
** last modification date for CruiseControl.lus (MM/DD/YYYY): 08/01/2006
*****************************************************************$*/

#include "Top_CruiseSpeedMgt_main.h"

#include "Top_CruiseSpeedMgt_extern.h"
#include "Top_CruiseSpeedMgt.h"

/* ============== */
/* INITIALISATION */
/* ============== */
```

```
void Top_CruiseSpeedMgt_init(_C_Top_CruiseSpeedMgt * _C_)
{
    CruiseSpeedMgt_init(&(_C_->_C_1_CruiseSpeedMgt));
    (_C_->_M_condact_0_Top_CruiseSpeedMgt) = true;
    (_C_->_M_init_Top_CruiseSpeedMgt) = true;
}

/* ========== */
/* MAIN NODE */
/* ========== */

void Top_CruiseSpeedMgt(_C_Top_CruiseSpeedMgt * _C_)
{
/*#code for node Top_CruiseSpeedMgt */
/* begin condact */
    if ((_C_->_I4_i_On)) {
/* occurrence number of the node CruiseSpeedMgt : 1 */
/* call to node not expanded CruiseSpeedMgt */
/* occurrence number of the node CruiseSpeedMgt : 1 */
        (_C_->_C_1_CruiseSpeedMgt._I0_i_Set) = (_C_->_I0_i_Set);
        (_C_->_C_1_CruiseSpeedMgt._I1_i_QuickAccel) =
            (_C_->_I1_i_QuickAccel);
        (_C_->_C_1_CruiseSpeedMgt._I2_i_QuickDecel) =
            (_C_->_I2_i_QuickDecel);
        (_C_->_C_1_CruiseSpeedMgt._I3_i_VehiculeSpeed) =
            (_C_->_I3_i_VehiculeSpeed);
        CruiseSpeedMgt(&(_C_->_C_1_CruiseSpeedMgt));
        (_C_->_O0_o_CruiseSpeed) =
            (_C_->_C_1_CruiseSpeedMgt._O0_o_CruiseSpeed);
        (_C_->_M_condact_0_Top_CruiseSpeedMgt) = false;
    } else {
        if (_C_->_M_init_Top_CruiseSpeedMgt) {
            (_C_->_O0_o_CruiseSpeed) = ZeroSpeed;
        }
    }
/* end condact */
    (_C_->_M_init_Top_CruiseSpeedMgt) = false;
/*#end code for node Top_CruiseSpeedMgt */
}



/*$************* SCADE_KCG KCG Version 4.2.0 (build i20) *************
** End of file Top_CruiseSpeedMgt.c
** End of generation (MM/DD/YYYY) : 08/01/2006 14:19:15
*************************************************************$*/
```

« CruiseSpeedMgt.c »

```
/*$************* SCADE_KCG KCG Version 4.2.0 (build i20) *************
** Command :
** l2C          CruiseControl.lus -node Top_CruiseSpeedMgt
**      -noexp @ALL@
**      -noassert
**      -const
**      -bitwise
**      -split
**      -no_copy_mem
** date of generation (MM/DD/YYYY): 08/01/2006 14:19:15
** last modification date for CruiseControl.lus (MM/DD/YYYY): 08/01/2006
*****************************************************************$*/

#include "Top_CruiseSpeedMgt_main.h"

#include "Top_CruiseSpeedMgt_extern.h"
#include "CruiseSpeedMgt.h"

/* ================ */
/* INITIALISATION */
/* ================ */

void CruiseSpeedMgt_init (_C_CruiseSpeedMgt *_C_)
{
(_C_->_M_init_0_Top_CruiseSpeedMgt) = true ;
}

/* ========== */
/* MAIN NODE */
/* ========== */

void CruiseSpeedMgt (_C_CruiseSpeedMgt *_C_)
{
Speed _L10_Top_CruiseSpeedMgt;
real _L13_Top_CruiseSpeedMgt;
real _L14_Top_CruiseSpeedMgt;
real _L0_Top_CruiseSpeedMgt;
real _L1_Top_CruiseSpeedMgt;
/*#code for node CruiseSpeedMgt */
if ((_C_->_M_init_0_Top_CruiseSpeedMgt))
{
_L10_Top_CruiseSpeedMgt = ZeroSpeed;
}
else
{
_L10_Top_CruiseSpeedMgt = (_C_->_O0_o_CruiseSpeed);
}
_L13_Top_CruiseSpeedMgt = (_L10_Top_CruiseSpeedMgt + SpeedInc);
_L14_Top_CruiseSpeedMgt = (_L10_Top_CruiseSpeedMgt - SpeedInc);
```

```
if (((_L14_Top_CruiseSpeedMgt >= SpeedMin) & (_C_->_I2_i_QuickDecel)))
{
_L0_Top_CruiseSpeedMgt = _L14_Top_CruiseSpeedMgt;
}
else
{
_L0_Top_CruiseSpeedMgt = _L10_Top_CruiseSpeedMgt;
}
if (((_L13_Top_CruiseSpeedMgt <= SpeedMax) & (_C_->_I1_i_QuickAccel)))
{
_L1_Top_CruiseSpeedMgt = _L13_Top_CruiseSpeedMgt;
}
else
{
_L1_Top_CruiseSpeedMgt = _L0_Top_CruiseSpeedMgt;
}
if ((_C_->_I0_i_Set))
{
(_C_->_O0_o_CruiseSpeed) = (_C_->_I3_i_VehiculeSpeed);
}
else
{
(_C_->_O0_o_CruiseSpeed) = _L1_Top_CruiseSpeedMgt;
}
(_C_->_M_init_0_Top_CruiseSpeedMgt) = false;
/*#end code for node CruiseSpeedMgt */
}



/*$************* SCADE_KCG KCG Version 4.2.0 (build i20) **************
** End of file CruiseSpeedMgt.c
** End of generation (MM/DD/YYYY) : 08/01/2006 14:19:15
*************************************************************$*/
```

ANNEXE II

EXPÉRIENCE D'UTILISATION

II.1 Énoncés de l'expérience

Les sections suivantes présentent les énoncés de l'expérience d'utilisation utilisées par les participants.

II.1.1 Énoncé principal

Voir pages 113 à 115

II.1.2 Énoncé Séance SCADE

Voir pages 116 à 119

II.1.3 Énoncé Séance C

Voir pages 120 à 122

Expérience d'utilisation Esterel SCADE - C

1. Objectif général

L'objectif principal de ce laboratoire est de vérifier, à l'aide d'une expérience de modélisation, laquelle des modélisations SCADE ou C permet d'être productive le plus rapidement. Cette expérience vise à simuler un cas industriel où un ingénieur se fait assigner à son premier ou à un nouveau projet.

Pour ce faire, chaque étudiant aura à effectuer des manipulations sur un design en SCADE et un design en C d'un système de régulation de vitesse. L'étudiant devra déterminer le temps nécessaire pour accomplir chacune des manipulations et par la suite comparer son travail à l'aide des deux designs. À la fin de chacune des séances de laboratoire, l'étudiant devra répondre à un questionnaire.

2. Description : Cruise Control

Le système de régulation de vitesse (ou Cruise Control) est un système embarqué dans une voiture qui maintient une vitesse constante sans à avoir à presser l'accélérateur ou la pédale de frein. Ce système possède une interface usager à partir de laquelle on peut configurer le régulateur (par exemple, régler la vitesse de croisière).

3. Spécifications du système

Vitesse du véhicule (Vehicule Speed) = Vitesse actuelle du véhicule
Vitesse de croisière (Cruise Speed) = Vitesse à laquelle le régulateur de vitesse est configuré

Le système se comporte comme suit :

- Les commandes de contrôle du Cruise Control sont « On » (activation), « Off » (arrêt), « Set » (réglage), « Resume » (reprise), « QuickAccel » (accélération rapide), « QuickDecel » (décélération rapide).
- Le système de régulation peut être utilisé si la vitesse du véhicule est plus grande que 30 km/h (*SpeedMin*) et plus petite que 150 km/h (*SpeedMax*).
- Lorsque le véhicule a atteint la vitesse désirée, le conducteur doit appuyer sur le bouton (en plus d'appuyer une deuxième fois pour le relâcher) « On » (ou « Set » si le mode du régulateur est déjà à REGUL_ON) pour sauvegarder la vitesse de croisière (Cruise Speed) et il doit arrêter de presser l'accélérateur ou la pédale de frein.
- Si le conducteur appuie sur l'accélérateur, la vitesse du véhicule augmentera temporairement et retournera à la vitesse de croisière lorsque le l'accélérateur sera relâché.
- Si aucun des boutons de contrôle (On, Off, Set ou Resume) ne sont appuyés et qu'on appuie sur l'accélérateur ou la pédale de frein, la vitesse du véhicule changera, mais la vitesse de croisière restera inchangée.
- Si la pédale de frein est pressée, la vitesse de croisière mémorisée peut être reprise en appuyant sur le bouton « Resume » (en plus d'appuyer une deuxième fois pour le relâcher).
- La vitesse de croisière peut être augmentée automatiquement (ou diminuée) de 5 km/h par seconde en appuyant sur le bouton « QuickAccel » (ou « QuickDecel »).
- Le système de régulation de vitesse peut-être arrêté en appuyant sur le bouton « Off »

Voici une description des deux modules principaux :

3.1 CRUISE STATE MANAGEMENT

Le module « Cruise State Management » détermine le mode du régulateur de vitesse (REGUL_ON, REGUL_OFF, REGUL_STDBY). Ces modes sont fixés à l'aide de la vitesse actuelle du véhicule, de la pédale de frein et d'accélération et des boutons de contrôle du régulateur. Notez que plus d'un mode peut être actif en même temps.

Entrées du module (type entre parenthèses):

- Accélérateur – *Accel* (percent)
- Frein – *Brake* (percent)
- Vitesse du Véhicule – *Speed* (speed)
- Bouton d'activation – *ON* (bool)
- Bouton d'arrêt – *OFF* (bool)
- Bouton de reprise – *RESUME* (bool)

Sorties du module:

- REGUL_ON/REGUL_OFF/REGUL_STDBY

Spécification détaillée

La pédale d'accélération est pressée si le pourcentage de l'entrée *Accel* est plus grand que 0.
Le frein est pressé si le pourcentage de l'entrée *Brake* est plus grand que 0.

La vitesse du véhicule est considérée comme valide si elle se situe à l'intérieur des limites permises, soit :

- La vitesse du véhicule est plus grande que *SpeedMin* km/h, et
- La vitesse du véhicule est plus petite que *SpeedMax* km/h.

Le régulateur de vitesse a trois modes possibles:

Regul_OFF:
Actif lorsque:
- À l'état initial du système
- La pédale de frein ou le bouton d'arrêt (Off) est pressé(e).

Regul_ON:
Actif lorsque:
- Le régulateur était arrêté (REGUL_OFF) et que le bouton On ou Resume est pressé, en autant que la vitesse du véhicule est valide et que l'accélérateur et le frein ne sont pas pressés
- Le régulateur était en mode REGUL_STDBY, la vitesse du véhicule est valide et la pédale d'accélération n'est pas pressée.

Regul_STDBY:
Actif lorsque:
- Le régulateur était arrêté (REGUL_OFF), et que le bouton On ou Resume est pressé, en autant que la vitesse du véhicule est valide, et que l'accélérateur est pressé et le frein n'est pas pressé
- Le régulateur était en marche (REGUL_ON), et que la vitesse du véhicule n'est plus valide ou que la pédale d'accélération est pressée ou que le frein est pressé.

Initialisation

À l'initialisation du système, *REGUL_OFF* est actif.

3.2 *CRUISE SPEED MANAGEMENT*

Le module « Cruise Speed Management » calcul la vitesse de croisière à l'aide de l'état du régulateur, la vitesse actuelle du véhicule et les commandes de contrôles qui sont pressées.

Entrées du module (type entre parenthèses):

- État du Régulateur
- Vitesse du Véhicule – *Speed* (speed)
- Bouton d'activation – *ON* (bool)
- Bouton de réglage – *SET* (bool)
- Bouton d'accélération rapide – *QUICKACCEL* (bool)
- Bouton de décélération rapide – *QUICKDECEL* (bool)

Sorties du module:

- Vitesse de croisière (speed)

Spécification détaillée

Le module « Cruise Speed management » est actif seulement et seulement si le mode du régulateur est *REGUL_ON* ou *REGUL_STDBY* (sinon, la vitesse de croisière ne change pas – c'est-à-dire la même que la précédente)

Si les boutons ON et SET sont pressés, la vitesse de croisière est égale à la vitesse du véhicule
Sinon,

- Si le bouton QUICKACCEL est pressé, la vitesse de croisière est augmentée de *SpeedInc* km/h, seulement si la nouvelle valeur est plus petite que la vitesse maximale *SpeedMax* km/h.
- Si le bouton QUICKDECEL est pressé, la vitesse de croisière est diminuée de *SpeedInc* km/h, seulement si la nouvelle valeur est plus grande que la vitesse minimale *SpeedMin* km/h.

Initialisation

À l'initialisation du système, la vitesse de croisière est 0 km/h.

4. Informations diverses et Travail à effectuer

Voir document SÉANCE SCADE (si vous faites la séance SCADE)

Voir document SÉANCE C (si vous faites la séance C)

5. Questionnaire et évaluation

Vous devez:

- Avoir complété les manipulations demandées (**1re et 2e séance**)
- Avoir cumulé le temps nécessaire pour compléter chacune des manipulations et noté les valeurs sur votre feuille de temps (**1re et 2e séance**)
- Avoir rempli le questionnaire spécifique à votre séance (C ou SCADE) (**1re et 2e séance**)
- *Avoir rempli le questionnaire final sur vos impressions* (**2e séance**)

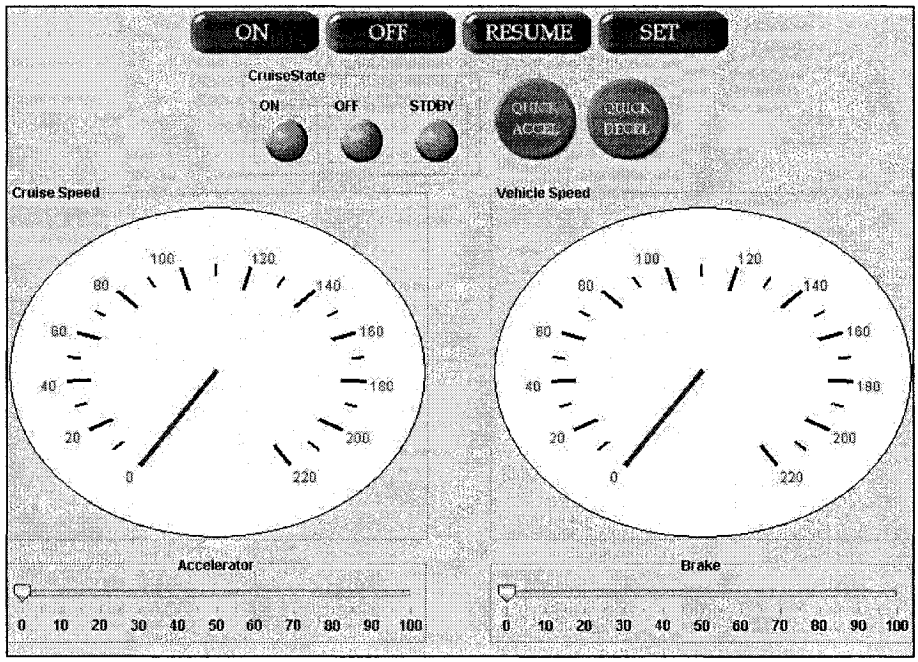
Expérience d'utilisation
Séance SCADE

1. Tutoriel

Avant de continuer plus loin. Veuillez faire le tutoriel sur SCADE. Ce tutoriel prendra environ 45 minutes. Il est **absolument nécessaire** de le compléter avant de commencer le projet.

Le Tutoriel (un document nommé : « Tutoriel SCADE 5.01.pdf ») vous a été envoyé avec les fichiers du laboratoire.

Notez le temps que vous avez pris pour compléter le tutoriel dans la feuille de temps à la fin de cet énoncé.



2. Informations diverses

Avant de commencer, faites l'extraction du projet « **CruiseControl_SCADE_Départ** » à partir de l'archive CruiseControl_SCADE_Départ.zip qui vous a été envoyée par courriel avec les fichiers du projet.
Dans le répertoire CruiseControl_SCADE_Départ/CruiseControl/, ouvrez à l'aide de SCADE le fichier de projet « **CruiseControl.vsw** »

Pour permettre la simulation, une librairie SCADE simulant le comportement d'une voiture (Car_Simple) vous a été fournie. Dans votre projet, le nœud « Cruise_System » permet de simuler votre système complet comprenant le nœud « CruiseControl » (implémentant le régulateur de vitesse) et le nœud Car_Simple, simulant la voiture.

Une interface JAVA est disponible pour ce projet. Cette interface vous aidera à simuler votre système en vous permettant d'interagir plus facilement avec celui-ci. Les boutons glissoirs de l'accélérateur et du frein vous permettront de simuler la pression appliquée sur le frein et l'accélérateur. Deux cadrans vous indiquent la vitesse de croisière et la vitesse du véhicule. Trois lumières vous donnent l'état dans lequel le système de régulation se trouve. Vous avez aussi six boutons simulant les commandes de contrôle ON – OFF – RESUME – SET – QUICKACCEL – QUICKDECEL.

Pour simuler votre système à l'aide de l'interface JAVA, allez dans le menu « Project », ensuite choisissez « Code Generation – Simulator » et « Set Active Configuration ». Dans la liste, sélectionnez « SimuJavaBeans » et cliquez sur Set Active et OK.

Pour démarrer le simulateur, faites un clic droit sur le nœud « Cruise_System » et cliquez sur « Simulate Node Cruise System ».

Vous pouvez l'essayer avant de commencer les manipulations...

3. Travail à effectuer

Vous devez comptabiliser le temps nécessaire pour compléter chacune des parties. Veuillez vous référer à la feuille de temps en annexe. Le temps débute dès que vous commencez à travailler sur une des manipulations, et finit dès que celle-ci a été testée et considérée comme « fonctionnant ».
Allez lire la feuille de temps avant de commencer les manipulations.

3.1. Correction de bogues

Deux bogues se sont introduits dans l'application Cruise Control.

Bogue #1
Il y a un problème avec les modes REGUL_STDBY et REGUL_ON. Dès que les freins sont pressés à partir d'un système en marche (avec une vitesse valide et l'accélérateur qui n'est pas pressé) le système se retrouve dans le mode REGUL_STDBY (ce qui est normal), mais aussitôt que les freins sont relâchés le système retourne dans le mode REGUL_ON, sans qu'on aille à appuyer sur le bouton Resume.

Pour vous aider, référez-vous à la spécification du « Cruise State Management » de l'énoncé du laboratoire.

→ *Notez votre temps sur la feuille de temps.*

Bogue #2
Lorsqu'on appuie sur le frein à partir d'un système en marche, il y a un problème avec les modes affichés par le système (c.-à-d. les lumières de l'interface java).

Pour vous aider, référez-vous à la spécification du « Cruise State Management » de l'énoncé du laboratoire.

→ *Notez votre temps sur la feuille de temps.*

3.2. Ajout d'une fonctionnalité

Le nœud « CruiseSpeedMgt » doit être complètement implémenté.

Pour ce faire, référez-vous à la spécification sur le « Cruise Speed Manager » de l'énoncé du laboratoire.

Lorsque vous l'aurez implémenté, veuillez tester votre système afin de vérifier son bon fonctionnement.

→ *Notez votre temps sur la feuille de temps.*

3.3. Intégration

Maintenant que les bogues sont corrigés et le module « CruiseSpeedMgt » implémenté et simulé, vous êtes prêt à intégrer le régulateur de vitesse dans le système de gestion des composants de la voiture. Le système de

gestion émule des opérations sur des composants de la voiture. Ce système fonctionne sous µC/OS-II, donc les opérations sont séparées en tâches (votre Régulateur de vitesse deviendra lui aussi une tâche). Pour connaître ces opérations, veuillez vous référer aux commentaires dans le code source du système.

Les prochaines sections vous donneront la marche à suivre pour faire l'intégration finale.

3.3.1 Génération de Code à partir de SCADE

Vous devez premièrement générer les fichiers C du CruiseControl et le wrapper µC/OS-II pour vous permettre de faire l'intégration dans le système de gestion.

Pour générer le code, vous devez :

1. Choisir la configuration « Standard » dans SCADE (cliquez sur PROJET – CODE GENERATOR SIMULATOR – SET ACTIVE CONFIGURATION et choisissez « Standard »). Cette configuration permettra de générer seulement les fichiers dont vous avez besoin pour intégrer le CruiseControl dans le système de gestion. En laissant la config de SimuJavaBeans, d'autres fichiers auraient été générés (qui ne vous n'aurait pas été utile).
2. Pour faire des modifications au wrapper µC/OS-II, allez à l'onglet MicroC/OS-II de la fenêtre « Settings » du « Code Generator-Simulator » (accessible en cliquant sur PROJET – CODE GENERATOR SIMULATOR – SETTINGS). Vous pouvez faire toutes les modifications que vous voulez (à l'exception du TaskName qui doit demeurer à la valeur par défaut « $(NodeName)_MicroC »).
3. Faites un clic droit sur le nœud « Cruise_System » et choisissez « Generate Node Cruise_System ». Vérifiez que la génération se complète sans erreurs.
4. Les fichiers générés se retrouveront dans le répertoire **\CruiseControl_SCADE_Départ\CruiseControl\Cruise_System_C\Standard**
   Avant de générer le code, il est conseillé d'effacer tous les fichiers déjà présents dans ce répertoire.

3.3.2 Ouverture du projet « Système de Gestion »

Le système de gestion est contenu dans le projet « **Systeme_Gestion_SCADE_Départ** ». L'archive du projet vous a été envoyé par courriel avec les autres fichiers du laboratoire (fichier **Systeme_Gestion_SCADE_Départ.zip**). Faites maintenant l'extraction du système de gestion.
Le projet Systeme_Gestion_SCADE_Départ est un projet Visual Studio 6.0. Pour l'ouvrir, faites un clic droit sur le fichier **CruiseControl_C.dsw**, choisissez « Open With » Visual Studio 6.0 (**il ne faut pas** utiliser Visual Studio .NET 2003).

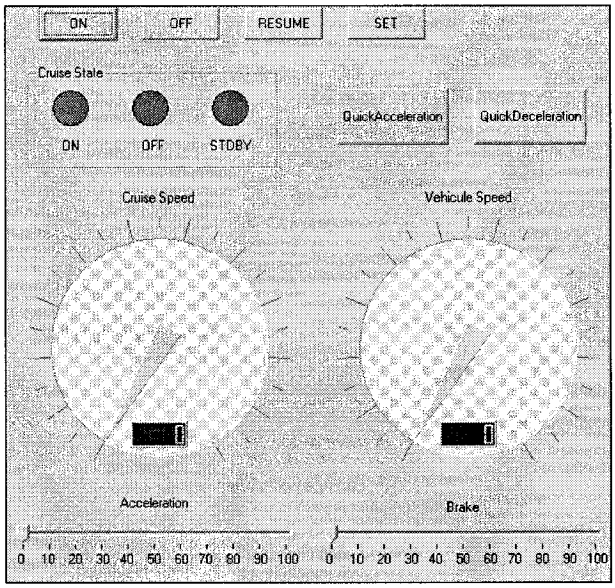
N.B. : Pour les gens qui ne sont pas familiers avec Visual Studio, sachez que tous les fichiers du projet sont accessibles sous l'onglet « FileView » de la fenêtre à gauche. Pour compiler le projet, appuyez sur F7. Pour l'exécuter, appuyez sur CTRL-F5.

3.3.3 Exécution et Interface Usager

Le système de gestion possède sa propre interface usager « Cruise Control ». Pour l'utiliser, accédez au répertoire **\Systeme_Gestion_SCADE_Départ \ui** et exécutez le programme **uiCruiseControl_C.exe**. Cette interface usager fonctionne exactement de la même manière que l'interface JAVABeans que vous avez utilisé dans SCADE.

Cette interface communique avec le Système de Gestion à l'aide d'un mécanisme de mémoire partagée. Donc, avant d'exécuter votre Système de Gestion, l'interface usager doit toujours être exécutée préalablement. Si vous oubliez de la démarrer avant d'exécuter votre projet (et de la redémarrer pour chaque nouvelle exécution), le message d'erreur suivant apparaîtra à la console :

```
Could not open file mapping object (2).
Avez-vous oublie de demarrer l'interface usager???
```



3.3.4 Manipulations

1. Copiez tous les fichiers avec l'extension .C et .h (**ET SEULEMENT CEUX-CI**) de votre répertoire de génération à votre projet « Système de Gestion ». Vous allez devoir écraser des fichiers déjà existants (ceci est normal, car le Systeme de Gestion contient déjà les fichiers du Cruise Control original, c'est-à-dire avec les bogues et le module « CruiseSpeedMgt » non implanté).
2. Dans le fichier principal du système de Gestion (main.c), vous allez devoir créer la tâche correspondante aux opérations du CruiseControl (à partir des fonctions créées dans le wrapper MicroC, voir fichier Cruise_System_MicroC.c).
3. Toujours dans le fichier principal du système de Gestion (main.c), vous allez devoir activer la tâche correspondante aux opérations du CruiseControl. Cette tâche sera activée par la tâche « TaskSharedMemory », qui s'occupe de communiquer avec l'interface usager. Voir le commentaire //TODO dans cette tâche qui vous indique où doit se faire l'activation. Cette activation se fera via les fonctions créées dans le wrapper MicroC, voir fichier Cruise_System_MicroC.c).
4. Testez votre système et assurez-vous d'une exécution conforme aux spécifications et identique à ce que vous avez fait avec SCADE.
5. Si vous devez faire des modifications dans votre Cruise Control, retournez à votre projet SCADE, faites vos modifications, testez celles-ci, refaites la génération de code, et réintégrez le tout au système de Gestion. ***Veuillez indiquer dans votre feuille de temps le nombre de fois où vous avez dû retourner à votre design SCADE pour faire des modifications***

→ *Notez votre temps sur la feuille de temps.*

Expérience d'utilisation
Séance C

1. Tutoriel

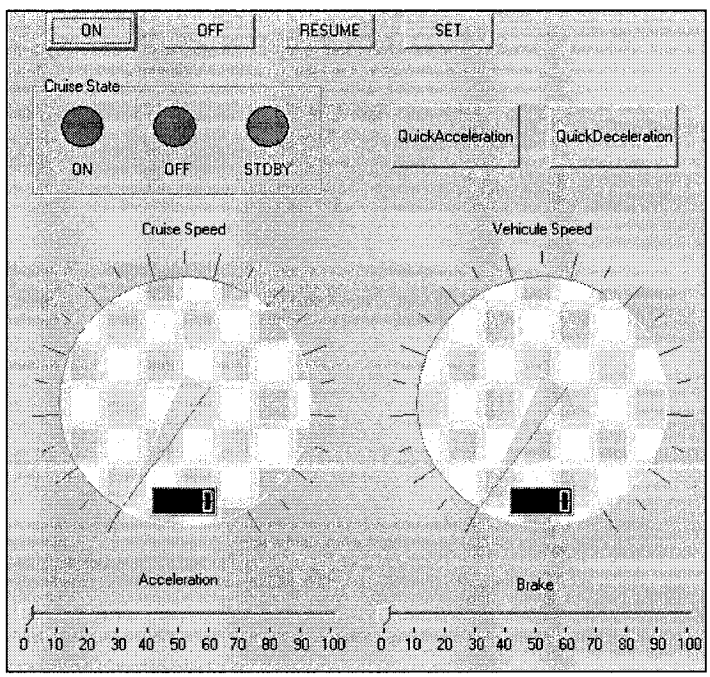
Il n'y a aucun tutoriel à faire pour cette séance.

2. Informations diverses

Avant de commencer, faites l'extraction du projet « **CruiseControl_C_Départ** » à partir de l'archive **CruiseControl_C_Départ.zip** qui vous a été envoyée par courriel avec les fichiers du projet.
Dans le répertoire \CruiseControl_C_Départ\CruiseControl\ ouvrez à l'aide de Visual Studio 6.0 le fichier de projet « **CruiseControl_C.dsw** ».

Le projet contient cinq fichiers. Dont voici une description de chacun :

- CruiseControl.c : Contiens l'implantation du système de régulation de vitesse. C'est dans ce fichier que vous ferez les manipulations demandées.
- Main.c : Contiens la boucle principale du programme permettant d'exécuter le système de régulation de vitesse et de communiquer avec l'interface usager via une zone de mémoire partagée.
- Car_simple.h : Contient l'API permettant de communiquer avec la librairie implémentant le comportement d'une voiture (nécessaire pour faire fonctionner le régulateur de vitesse).
- CruiseControl.h : Contiens les prototypes des fonctions du CruiseControl ainsi que les structures de données utilisées.
- Types.h : Contient les types utilisés par la librairie Car Simple.



Une interface usager est disponible pour ce projet. Cette interface vous aidera à simuler votre système en vous permettant d'interagir plus facilement avec celui-ci. Les boutons glissoirs de l'accélérateur et du frein vous permettront de simuler la pression appliquée sur le frein et l'accélérateur. Deux cadrans vous indiquent la vitesse de croisière et la vitesse du véhicule. Trois lumières vous donnent l'état dans lequel le système de régulation se trouve. Vous avez aussi six boutons simulant les commandes de contrôle ON – OFF – RESUME – SET – QUICKACCEL – QUICKDECEL.

Pour l'utiliser, accédez au répertoire **\CruiseControl_C_Départ\ui** et exécutez le programme **uiCruiseControl_C.exe**.
Cette interface communique avec le régulateur de vitesse à l'aide d'un mécanisme de mémoire partagée. Donc, avant d'exécuter votre régulateur, l'interface usager doit toujours être exécutée préalablement. Si vous oubliez de la démarrer avant d'exécuter votre projet (et de la redémarrer pour chaque nouvelle exécution), le message d'erreur suivant apparaîtra à la console :

```
Could not open file mapping object (2).
Avez-vous oublie de demarrer l'interface usager???
```

3. Travail à effectuer

Vous devez comptabiliser le temps nécessaire pour compléter chacune des parties. Veuillez vous référer à la feuille de temps en annexe. Le temps débute dès que vous commencez à travailler sur une des manipulations, et finit dès que celle-ci a été testée et considérée comme « fonctionnant ».
Allez lire la feuille de temps avant de commencer les manipulations.

3.1. Correction de bogues

Deux bogues se sont introduits dans l'application Cruise Control.

Bogue #1
Il y a un problème avec les modes REGUL_STDBY et REGUL_ON. Dès que les freins sont pressés à partir d'un système en marche (avec une vitesse valide et l'accélérateur qui n'est pas pressé) le système se retrouve dans le mode REGUL_STDBY (ce qui est normal), mais aussitôt que les freins sont relâchés le système retourne dans le mode REGUL_ON, sans qu'on aille à appuyer sur le bouton Resume.

Pour vous aider, référez-vous à la spécification du « Cruise State Management » de l'énoncé du laboratoire.

→ *Notez votre temps sur la feuille de temps.*

Bogue #2
Lorsqu'on appuie sur le frein à partir d'un système en marche, il y a un problème avec les modes affichées par le système (c.-à-d. les lumières de l'interface usager).

Pour vous aider, référez-vous à la spécification du « Cruise State Management » de l'énoncé du laboratoire.

→ *Notez votre temps sur la feuille de temps.*

3.2. Ajout d'une fonctionnalité

Le nœud « CruiseSpeedMgt » doit être complètement implémenté.
Pour ce faire, référez-vous à la spécification sur le « Cruise Speed Manager » de l'énoncé du laboratoire.
Lorsque vous l'aurez implémenté, veuillez tester votre système afin de vérifier son bon fonctionnement.
→ *Notez votre temps sur la feuille de temps.*

3.3. Intégration

Maintenant que les bogues sont corrigés et le module « CruiseSpeedMgt » implémenté et simulé, vous êtes prêt à intégrer le régulateur de vitesse dans le système de gestion des composants de la voiture. Le système de gestion émule des opérations sur des composants de la voiture. Ce système fonctionne sous µC/OS-II, donc les opérations sont séparées en tâches (votre Régulateur de vitesse deviendra lui aussi une tâche). Pour connaître ces opérations, veuillez vous référer aux commentaires dans le code source du système

Les prochaines sections vous donneront la marche à suivre pour faire l'intégration finale.

3.3.1 Ouverture du projet « Système de Gestion »

Le système de gestion est contenu dans le projet « **Systeme_Gestion_C_Départ** ». L'archive du projet vous a été envoyé par courriel avec les autres fichiers du laboratoire (fichier **Systeme_Gestion_C_Départ.zip**). Faites maintenant l'extraction du système de gestion.
Le projet « **Systeme_Gestion_C_Départ** » est aussi un projet Visual Studio 6.0. Pour l'ouvrir, faites un clic droit sur le fichier **CruiseControl_C.dsw**, choisissez « Open With » Visual Studio 6.0

3.3.2 Exécution et Interface Usager

Le système de gestion possède sa propre interface usager « Cruise Control ». Pour l'utiliser, accédez au répertoire **\Systeme_Gestion_C_Départ\ui** et exécutez le programme **uiCruiseControl_C.exe**. Cette interface usager fonctionne exactement de la même manière que l'interface que vous avez utilisée avec le projet du régulateur de vitesse.

Cette interface communique avec le Système de Gestion à l'aide d'un mécanisme de mémoire partagée. Donc, avant d'exécuter votre Système de Gestion, l'interface usager doit toujours être exécutée préalablement. Si vous oubliez de la démarrer avant d'exécuter votre projet (et de la redémarrer pour chaque nouvelle exécution), le message d'erreur suivant apparaîtra à la console :

```
Could not open file mapping object (2).
Avez-vous oublie de demarrer l'interface usager???
```

3.3.3 Manipulations

1. Copiez les fichiers CruiseControl.C et CruiseControl.h (**ET SEULEMENT CEUX-CI**) de votre projet « Régulateur de vitesse » à votre projet « Système de Gestion ».
2. Dans le fichier principal du système de Gestion (main.c), vous allez devoir créer la tâche correspondante aux opérations du CruiseControl (TaskCruiseControl).
3. Vous allez devoir maintenant implémenter la tâche TaskCruiseControl pour qu'elle fasse exactement le même travail que la fonction CruiseSystem() dans le projet du Régulateur de vitesse. Cette tâche devra attendre un sémaphore avant de pouvoir s'exécuter.
4. Toujours dans le fichier principal du système de Gestion (main.c), vous allez devoir activer TaskCruiseControl (à l'aide d'un sémaphore). Cette tâche sera activée par la tâche « TaskSharedMemory », qui s'occupe de communiquer avec l'interface usager. Voir le commentaire //TODO dans cette tâche qui vous indique où doit se faire l'activation.
5. Testez votre système et assurez-vous d'une exécution conforme aux spécifications.
6. Si vous devez faire des modifications dans votre Cruise Control, retournez à votre projet original du Régulateur de vitesse, faites vos modifications, testez celles-ci et réintégrez-le tout au système de Gestion. ***Veuillez indiquer dans votre feuille de temps le nombre de fois où vous avez dû retourner à votre design Régulateur de Vitesse pour faire des modifications***

→ *Notez votre temps sur la feuille de temps.*

II.2 Résultats de l'expérience

Voici les tableaux de résultats bruts pour chacun des programmeurs pour chacune des séances. Veuillez noter que les programmeurs ayant le numéro 1 à 17 ont débuté par la séance SCADE et les numéros 21 à 36 par la séance C. Ces deux groupes ont ainsi inversé leur rôle à la deuxième séance de laboratoire. Le programmeur #34 a été retiré des résultats, car celui-ci n'a pas complété sa deuxième séance.

Tableau II.1 Résultats de la séance SCADE (en minutes)

Programmeur	Bogue #1	Bogue #2	CruiseSpeedMgt	Intégration
1	33	35	118	95
2	30	3	60	20
3	30	30	120	30
4	60	5	170	150
5	20	20	150	15
6	100	60	130	90
7	120	10	140	20
8	22	53	57	15
9	28	10	43	45
10	15	7	62	18
11	10	15	80	15
12	45	11	73	37
13	35	10	180	30
14	15	35	65	30
15	30	30	75	60
16	120	2	10	10
17	20	5	61	28
21	13	5	45	12
22	10	2	100	130
23	15	10	120	60
24	25	10	60	10
25	3	3	38	5
26	12	3	32	10
27	5	5	65	10
28	15	3	47	20
29	8	10	96	12
30	10	5	25	20
31	8	2	35	20
32	15	15	180	120
33	7	7	25	20
35	15	16	30	10
36	9	6	19	6
TOTAL:	**903**	**443**	**2511**	**1173**

Tableau II.2 Résultats de la séance C (en minutes)

Programmeur	Bogue #1	Bogue #2	CruiseSpeedMgt	Intégration
1	17	15	45	45
2	5	2	40	60
3	15	15	110	30
4	15	2	30	60
5	10	20	70	120
6	20	10	40	90
7	10	3	110	320
8	5	4	35	25
9	23	12	53	65
10	3	8	15	25
11	10	20	65	25
12	2	2	69	28
13	5	7	50	40
14	7	3	60	95
15	5	5	20	15
16	20	1	25	30
17	11	1	13	50
21	150	10	40	40
22	35	4	47	110
23	120	30	10	360
24	45	15	120	30
25	10	10	40	20
26	2	5	31	20
27	20	10	50	25
28	5	9	20	70
29	22	31	58	10
30	5	5	30	20
31	20	5	32	60
32	90	15	150	180
33	90	10	180	90
35	20	15	75	120
36	42	7	36	23
TOTAL:	**859**	**311**	**1777**	**2301**

Tableau II.3 Résultats comparatifs (en minutes)

Programmeur	Total Séance C[1]	Total Séance SCADE[1]	Différence[2]	Rang (Wilcoxon)[3]
1	213	90	**-123**	6
2	80	100	**20**	24
3	150	140	**-10**	28
4	320	90	**-230**	2
5	165	190	**25**	21
6	220	135	**-85**	9
7	160	430	**270**	1
8	72	60	**-12**	27
9	88	118	**30**	18,5
10	80	43	**-37**	14,5
11	95	90	**-5**	30,5
12	110	97	**-13**	26
13	210	90	**-120**	7
14	95	155	**60**	12
15	135	35	**-100**	8
16	20	55	**35**	16
17	89	63	**-26**	20
21	57	80	**23**	22,5
22	230	157	**-73**	11
23	180	370	**190**	4
24	70	150	**80**	10
25	43	60	**17**	25
26	42	51	**9**	29
27	75	75	**0**	-
28	67	90	**23**	22,5
29	108	68	**-40**	13
30	45	50	**5**	30,5
31	55	92	**37**	14,5
32	300	330	**30**	18,5
33	45	270	**225**	3
35	40	195	**155**	5
36	25	59	**34**	17

[1] Les totaux n'incluent pas les temps cumulés pour corriger les bogues
[2] (Temps de la séance SCADE) - (Temps de la séance C)
[3] Voir section II.2.1.2 sur le test de Wilcoxon

II.2.1 Analyses statistiques

Cette sous-section présente les analyses statistiques réalisées à partir des résultats de l'expérience. Le but premier de ces analyses est d'essayer d'infirmer l'hypothèse nulle de façon statistique. Trois tests seront alors présentés soient : le Test-T pour échantillons appariés, le test Wilcoxon et le test signé. Le Test-T est un test paramétrique et les deux autres sont non paramétriques. Les tests paramétriques ont une puissance plus grande que les tests non paramétriques, dans le sens où ceux-ci ont besoin moins de points d'expérimentations pour trouver une tendance et sont généralement plus robustes aux déviations par rapport aux préconditions.

II.2.1.1 Test-T pour échantillons appariés

Le Test-T pour échantillons appariés ([54] p:101) est utilisé quand deux échantillons tirés de manipulations répétées sont comparés. Dans notre cas, les deux échantillons consistent en le projet de la séance SCADE et le projet de la séance C. Le test examine la performance des sujets de l'expérience qui ont travaillé sur les deux séances.

Le Test-T se défini comme suit:

Entrées: Échantillons en paire (x_1, y_1), (x_2, y_2), ..., et (x_n, y_n), où n est le nombre d'échantillons.

Hypothèse nulle (H_0): $\mu_d = 0$ où $d_i = x_i - y_i$ (c'est-à-dire que la moyenne des différences anticipée est égale à 0).

Calculs: $t_0 = \dfrac{\bar{d}}{S_d/(\sqrt{n})}$, où $S_d = \sqrt{\dfrac{\sum_{i=1}^{n}(d_i - \bar{d})^2}{n-1}}$, ($\bar{d}$ est la moyenne des différences).

Critères: En prévoyant une moyenne des différences plus grande que 0 ($\mu_d > 0$, c'est-à-dire que les temps de la séance C sont plus grand que ceux de la séance

SCADE), l'hypothèse nulle (H_0) peut-être rejetée si $t_0 > t_{\alpha,n-1}$. Ici, $t_{\alpha,n-1}$ est le α pourcentage de points plus élevés de la distribution t avec $n-1$ degrés de liberté. α est en d'autres termes la probabilité de rejeter l'hypothèse nulle quand celle-ci est en fait vraie.

À partir des résultats de l'expérience, les valeurs suivantes peuvent être calculées :

Moyenne des différences: $\bar{d} = 12{,}313$

Variance des différences: $S_d = 99{,}335$

Test-T: $t_0 = 0{,}701$

Critère de décision: $t_{0{,}05,31} = 2{,}042$

Décision: L'hypothèse nulle ne peut pas être rejetée car $t_0 \not> t_{0{,}05,31}$

La valeur du critère de décision (probabilité de 5% de mauvaise décision concernant l'hypothèse nulle) a été prise au Tableau A1 de [54].

II.2.1.2 Test Wilcoxon

Le test Wilcolxon ([54] p:103 et [49] p:87) est une alternative au Test-T. La seule obligation du test est qu'il soit possible de déterminer quelles mesures d'une paire de mesures est la plus grande et qu'il soit possible de classer les différences par rangs. Dans notre cas, plus la différence est grande et plus le rang est élevé (donc les différences ont été classées en ordre décroissant).

Le test Wilcoxon se défini comme suit:

Entrées: Échantillons en paire (x_1, y_1), (x_2, y_2), ..., et (x_n, y_n), où n est le nombre d'échantillons.

Hypothèse nulle (H_0): Si toutes les différences ($d_i = x_i - y_i$) sont ordonnées (1, 2, 3,...) en considérant seulement leur valeur absolue, la somme des rang des différences positives est égale à la somme des rang de différences négatives.

Calculs: T^+ est calculé comme étant la somme des rangs des d_i positifs et T^- est calculé comme étant la somme des rangs des d_i négatifs. Les valeurs $d_i = 0$ sont enlevées de l'échantillon. Par la suite, z doit être calculé comme suit: $z = \frac{T^+ - \mu_{T^+}}{\sigma_{T^+}} = \frac{T^+ - N(N+1)/4}{\sqrt{N(N+1)(2N+1)/24}}$

Critères: H_0 est rejeté si la valeur z calculée correspond à une probabilité p qui est plus petite ou égale à une valeur α. La valeur de p peut être trouvé au Tableau A de l'annexe de [49]

À partir des résultats de l'expérience (voir Tableau II.2 pour les rangs), les valeurs suivantes peuvent être calculées (l'échantillon du sujet #27 a été retiré car $d_{27} = 0$):

Somme des rangs positifs: $T^+ = 294$

Somme des rangs négatifs: $T^- = 202$

Critère de décision: $z = 0{,}90144$ -> $p = 0{,}1814$

Alpha: $\alpha = 0{,}05$ (5%)

Décision: L'hypothèse nulle ne peut pas être rejetée car $p \nless 0,05$

II.2.1.3 Test signé

Le test signé ([54] p:104), basé sur le test binomial, est aussi une alternative au Test-T. Le test signé est basé sur le signe des différences. Généralement, il n'est pas nécessaire de faire le test de Wilcoxon quand il est possible de montrer une tendance avec le test signé.

Le test signé se défini comme suit:

Entrées: Échantillons en paire (x_1, y_1), (x_2, y_2), ..., et (x_n, y_n), où n est le nombre d'échantillons.

Hypothèse nulle (H_0): Si P(+) = P(-), où + et - représente les deux évènements où $x_i > y_i$ et $x_i < y_i$

Calculs: Faire la sommation du nombre de différences ($d_i = x_i - y_i$) positives et du nombre de différences négatives. Les échantillons où $d_i = 0$ sont retirés du test. Par la suite, il faut calculer $p = \frac{1}{2^N} \sum_{i=0}^{n} \binom{N}{i}$, où N est le nombre total de signes et n le nombre total de signes le plus rare.

Critères: Si P(-) < P(+), H_0 est rejeté si $p < \alpha$ est que le signe - est le plus rare dans l'échantillon.

À partir des résultats de l'expérience, les valeurs suivantes peuvent être calculées (l'échantillon du sujet #27 a été retiré car $d_{27} = 0$) :

Somme des signes positifs: $P(+) = 18$

Somme des signes négatifs: $P(-) = 13$

Critère de décision: $p = 0{,}2365$

Alpha: $\alpha = 0{,}05$ (5%)

Décision: L'hypothèse nulle ne peut pas être rejetée car $p \not< 0,05$

ANNEXE III

OBJECTIFS DE VÉRIFICATION

Cette annexe présente les objectifs de vérification tels que décrits dans la DO-178B ainsi qu'un glossaire relatif à certains termes utilisés dans la norme.

III.1 Tables d'objectifs

Cette section présente les tables d'objectifs de vérification de la DO-178B et l'impact potentiel de l'utilisation de SCADE. Ces impacts peuvent être :

- **Efficacité**: L'activité de vérification est beaucoup plus efficace
- **Éliminé**: L'activité de vérification peut être éliminée
- **Automatisé**: L'activité de vérification peut être automatisée
- **Aucun**: Aucun impact significatif

Il faut noter que chacun de ces impacts s'applique aux activités de vérification réalisées sur des modèles faits et générés à partir de SCADE.

Le nom des tables est le même que les tables d'objectifs à l'annexe A de la DO-178B [46]. Pour avoir une explication de chacun des impacts pour chacun des objectifs [15].

III.2 Glossaire

Ces définitions sont tirées directement du glossaire de la DO-178B.

Tableau III.1 Vérification des produits de la conception logiciel(Table A-4 de la DO-178B)

	Objectifs	Impact
1	Spécifications de bas-niveau sont en conformité avec les spécifications de haut-niveau	Efficacité
2	Spécifications de bas-niveau sont précises et cohérentes	Efficacité
3	Spécifications de bas-niveau sont compatibles avec la machine cible	Efficacité
4	Spécifications de bas-niveau sont vérifiables	Éliminé
5	Spécifications de bas-niveau sont conformes aux standards	Automatisé
6	Spécifications de bas-niveau sont traçables jusqu'aux spécifications de haut-niveau	Efficacité
7	Les algorithmes sont précis	Efficacité
8	L'architecture logicielle est compatible aux spécifications de haut-niveau	Efficacité
9	L'architecture logicielle est cohérente	Efficacité
10	L'architecture logicielle est compatible avec la machine cible	Efficacité
11	L'architecture logicielle est vérifiable	Efficacité
12	L'architecture logicielle est conforme aux standards	Automatisé
13	L'intégrité de l'architecture logicielle est confirmée	Aucun

Tableau III.2 Vérification des produits du codage et de l'intégration du logiciel(Table A-5 de la DO-178B)

	Objectifs	Impact
1	Le code source est en conformité avec les spécifications de bas-niveau	Éliminé
2	Le code source est conforme à l'architecture du logiciel	Éliminé
3	Le code source est vérifiable	Éliminé
4	Le code source est conforme aux standards	Éliminé
5	Le code source est traçables vers les spécifications de bas-niveau	Éliminé
6	Le code source est précis et cohérent	Éliminé
7	La sortie du processus d'intégration logicielle est complète et correctte	Aucun

Tableau III.3 Test des produits de l'intégration(Table A-6 de la DO-178B)

	Objectifs	Impact
1	Le code objet exécutable est conforme aux spécifications de haut-niveau	Efficacité
2	Le code objet exécutable est robuste vis-à-vis des spécifications de haut-niveau	Efficacité
3	Le code objet exécutable est conforme aux spécifications de bas-niveau	Efficacité
4	Le code objet exécutable est robuste vis-à-vis des spécifications de bas-niveau	Efficacité
5	Le code objet exécutable est compatible à la machine cible	Efficacité

Tableau III.4 Vérification des produits de la vérification(Table A-7 de la DO-178B)

	Objectifs	Impact
1	Les procédures de test sont correctes	
2	Les résultats des tests sont corrects et les écarts expliqués	Efficacité
3	La couverture de test des spécifications de haut-niveau est assurée	Efficacité
4	La couverture de test des spécifications de bas-niveau est assurée	Éliminé
5	La couverture structurelle des conditions/décisions modifiés est assurée	Éliminé
6	La couverture structurelle (couverture des décisions) est assurée	Éliminé
7	La couverture structurelle (couverture des directives) est assurée	Éliminé
8	La couverture structurelle (couplage contrôle/donnée) est assurée	Efficacité

Code désactivé

Code objet exécutable (ou données) qui, par conception, soit (a) n'est pas prévu pour être exécuté (ou utilisées), par exemple une partie d'un composant logiciel développé antérieurement, soit (b) n'est exécuté (ou utilisées) que dans certaines configurations de l'environnement de la machine cible, par exemple un code activable selon la valeur d'une broche matérielle ou en fonction d'options programmées par logiciel.

Code mort

Code objet exécutable (ou données) qui, en raison d'une erreur de conception, ne peut être exécuté (ou utilisées) dans une configuration opérationnelle de l'environnement de la machine cible et qui n'est pas traçable vers une spécification du système ou du logiciel. Ne sont pas considérés les identificateurs incorporés au logiciel.

Couverture des décisions

Chaque point d'entrée et de sortie du programme a été mis en oeuvre au moins une fois et chaque décision du programme a produit au moins une fois tous les résultats possibles.

Couverture des instructions

Chaque instruction du programme a été mise en oeuvre au moins une fois.

Couverture modifiée de condition/décision

Chaque point d'entrée et de sortie du programme a été mis en oeuvre au moins une fois, chaque condition d'un point de décision du programme a produit au moins une fois tous les résultats possibles, chaque décision du programme a produit au moins une fois tous les résultats possibles, et on a démontré que chaque condition dans une décision affecte de manière indépendante le résultat de cette décision. On démontre qu'une condition affecte de manière

indépendante le résultat d'une décision en ne faisant varier que cette condition et en laissant inchangées toutes les autres conditions.

Validation

Le processus assurant que les spécifications sont correctes et qu'elles sont complètes. Le processus du cycle de vie du système peut utiliser les spécifications du logiciel et les exigences dérivées pour la validation du système.

Vérification

L'évaluation des produits d'un processus dans le but de garantir leur exactitude et leur cohérence vis à vis des données d'entrée et des règles appliquées à ce processus.

ANNEXE IV

MODULE GESTIONNAIRE DE MESSAGES

IV.1 Glossaire

Tableau IV.1 Glossaire des données utilisées par le module Gestionnaire de messages

Données	Définition
Mission HAT	Mission Height above point of impact
AGL	Actual altitude
TAS	True Air Speed
IAS	Indicated Air Speed
CRP	CARP Waypoint (Green Light)
XTE	Extended Trailing Edge Waypoint (Red Light)
TE	Trailing Edge
IP	Identification Point
PI	Point of Impact
VERT DIST	Vertical Distance
QNH	Baro Altitude Correction

IV.2 Spécifications Système Textuelles

Cette section présente les Spécification Système Textuelle du module Gestionnaire de message. Ce module est en fait une partie de l'application CARP. Il est donné à titre d'exemple pour illustrer une application de la méthodologie présentée chapitre 4.

Chacune des SST possède un numéro d'identification distinctif.

[SST#1]

CARP Alert and Advisory Messages :

The OP CSCI shall generate the following CARP alert messages when the specified condition occurs:

Tableau IV.2 CARP Alert and Advisory Messages SST#1 Table

ID	Message	Condition
SST#1a	CARP ALT NOT MET	The actual altitude (AGL) is not within 150 feet of the Mission HAT within 60 seconds of reaching the CARP Green Light (CRP) waypoint.
SST#1b	CARP TAS NOT MET	The TAS is not within 20 knots of the TAS derived from the Mission IAS within 60 seconds of reaching the CARP Green Light (CRP) waypoint.

[SST#2]

In the presence of the NO CARP SOLUTION alert message, the OP CSCI shall not display the following status advisory messages:

- 20 MIN TO CARP
- 10 MIN TO CARP
- 1 MIN TO CARP
- 5 SEC TO CARP
- GREEN LIGHT
- RED LIGHT

[SST#3]

The OP CSCI shall remove and only remove the display of the CARP ALT NOT MET, CARP TAS NOT MET, NO CARP SOLUTION, messages if the condition(s) which triggered them is removed, or the CARP procedure is terminated.

[SST#4]

CARP Status Advisory message :

The OP CSCI shall generate the following CARP status advisory messages when the specified condition occurs:

Tableau IV.3 CARP Status Advisory Messages SST#4 Table

ID	Message	Condition
SST#4a	20 MIN TO CARP	20 minutes remaining prior to reaching the CRP waypoint.
SST#4b	10 MIN TO CARP	10 minutes remaining prior to reaching the CRP waypoint.
SST#4c	1 MIN TO CARP	1 minute remaining prior to reaching the CRP waypoint.
SST#4d	5 SEC TO CARP	5 seconds remaining prior to reaching the CRP waypoint.
SST#4e	GREEN LIGHT	CARP (Green Light) waypoint has been reached.
SST#4f	RED LIGHT	XTE (Red Light) waypoint has been reached.

[SST#5]

CARP Data Advisory message :

The OP CSCI shall generate the following CARP data entry advisory messages when the specified condition occurs:

Tableau IV.4 CARP Data Advisory Messages SST#5 Table

ID	Message	Condition
SST#5a	!TE TO IP OUT OF RANGE	The entered TE or IP position causes the distance from the TE to IP waypoints to be greater than 50NM.
SST#5b	!TE TO PI OUT OF RANGE	The entered TE or PI position causes the distance from the TE to PI waypoints to be greater than 20NM.
SST#5c	!IP TO PI OUT OF RANGE	The entered PI or IP position causes the distance from the PI to IP waypoints to be greater than 30NM.
SST#5d	!VERT DIST EXCEEDS HAT	The current VERT DIST value exceeds current HAT value. This advisory message is generated whenever this condition is met.

[SST#6]

Conditions for NO CARP SOLUTION :

Comment: The CARP procedure is by default achievable and becomes unachievable if one of the following events occurs:

Tableau IV.5 Conditions for NO CARP SOLUTION SST#6 Table

ID	Condition
SST#6a	The "CARP ACTIVE" discrete is on and the Baro Altitude is not available.
SST#6b	The "CARP ACTIVE" discrete is on and the temperature is not available.
SST#6c	The "CARP ACTIVE" discrete is on and the Baro Altitude Correction (QNH) is not available.
SST#6d	Less than 1 minute remains to the CRP waypoint and the actual Altitude (AGL) is smaller than the Vertical Distance (Ballistic Parameters).
SST#6e	Less than 1 minute remains to the CRP waypoint and the actual Altitude (AGL) differs from the Mission HAT by 1000ft.
SST#6f	Less than 1 minute remains to the CRP waypoint and the actual TAS differs from the TAS derived from the Mission IAS by 50 knots.

From being unachievable, the CARP solution becomes achievable when any of the following conditions occurs:

Tableau IV.6 Conditions for recovery from NO CARP SOLUTION SST#6 Table

ID	Condition
SST#6g	The solution was no longer achievable due to an unavailability of Baro Altitude, Temperature, or Altitude Correction (QNH), and the condition is restored prior to 1 minute before the first CARP procedure waypoint (IP or TP).
SST#6h	The solution was no longer achievable due to the actual AGL altitude or TAS and the condition is restored prior to 30 seconds before the CARP Green Light (CRP) waypoint.

Otherwise, the CARP solution remains unachievable.

[SST#7]

« CARP Active » Annunciator

The OP CSCI shall provide a « CARP Active » discrete output that will be set to true if and only if:

1. the aircraft is airbone and
2. 20 minutes remain to the CARP green light (CRP)

[SST#8]

« At CARP » Annunciator

The OP CSCI shall provide an "At CARP" discrete output that:

Tableau IV.7 « At CARP » Annunciator behaviour SST#8 Table

ID	Behaviour
SST#8a	is held to false at 5 seconds before CRP.
SST#8b	is cleared when the XTE (red light) point is sequenced.

[SST#9]

Calculation of the distance between two waypoints (in degree coordinates)

The distance (in nm.) is calculated from the formula of the « Great Circle Distances »

$$\cos D = (\sin a)(\sin b) + (\cos a)(\cos b)(\cos P)$$

where:

« **D** » is the angular distance between points A and B
« **a** » is the latitude of point A (North = positive value, South = negative value)
« **b** » is the latitude of point B (West = positive value, East = negative value)
« **P** » is the longitudinal difference between points A and B

The distance in nm. is obtained by multiplying the angle in degrees by the average distance of 1 degree of latitude in nm (1 Deg. = 59.935 nm).

IV.3 Tests

[Tests_SST#1]

Les tests doivent être effectués sur le module CARP_Alert_Msg

Tableau IV.8 SST#1a_T1

[SST#1a_T1]	Vérifie que le message CARP ALT NOT MET n'est pas généré lorsque la différence entre l'altitude AGL et la hauteur HAT est plus grande que 150 pieds et que le temps avant d'atteindre le point CRP est plus grand que 60 secondes.
Entrées :	i_rAGL = [a] 1000, [b] 700 i_rHAT = [a] 700, [b] 1000 i_TO_GREEN_LIGHT = [a] 75, [b] 75 i_rTAS = 0.0 i_rIAS = 0.0
Sorties anticipées :	o_msgCARP_ALT_NOT_MET = [a,b] false o_msgCARP_TAS_NOT_MET = $\oslash$

Tableau IV.9 SST#1a_T2

[SST#1a_T1]	Vérifie que le message CARP ALT NOT MET est généré lorsque la différence entre l'altitude AGL et la hauteur HAT est plus grande que 150 pieds et que le temps avant d'atteindre le point CRP est plus petit ou égal à 60 secondes.
Entrées :	i_rAGL = [a] 1000, [b] 700, [c] 1000, [d] 700 i_rHAT = [a] 700, [b] 1000, [c] 700, [d] 1000 i_TO_GREEN_LIGHT = [a] 45, [b] 45, [c] 60, [d] 60 i_rTAS = 0.0 i_rIAS = 0.0
Sorties anticipées :	o_msgCARP_ALT_NOT_MET = [a,b,c,d] true o_msgCARP_TAS_NOT_MET = $\oslash$

Tableau IV.10 SST#1a_T3

[SST#1a_T3]	Vérifie que le message CARP ALT NOT MET n'est pas généré lorsque la différence entre l'altitude AGL et la hauteur HAT est plus petite ou égale à 150 pieds et que le temps avant d'atteindre le point CRP est plus grand que 60 secondes.
Entrées :	i_rAGL = [a] 1000, [b] 900, [c] 1000, [d] 850 i_rHAT = [a] 900, [b] 1000, [c] 850, [d] 1000 i_TO_GREEN_LIGHT = [a] 45, [b] 45, [c] 60, [d] 60 i_rTAS = 0.0 i_rIAS = 0.0
Sorties anticipées :	o_msgCARP_ALT_NOT_MET = [a,b,c,d] false o_msgCARP_TAS_NOT_MET = ⊘

Tableau IV.11 SST#1a_T4

[SST#1a_T4]	Vérifie que le message CARP ALT NOT MET n'est pas généré lorsque la différence entre l'altitude AGL et la hauteur HAT est plus petite ou égale à 150 pieds et que le temps avant d'atteindre le point CRP est plus petit ou égal à 60 secondes.
Entrées :	i_rAGL = [a,c,e,g] 1000, [b,d,f,h] 900 i_rHAT = [a,c,e,g] 900, [b,d,f,h] 1000 i_TO_GREEN_LIGHT = [a,b] 45, [c,d] 60, [e,f] 45, [g,h] 60 i_rTAS = 0.0 i_rIAS = 0.0
Sorties anticipées :	o_msgCARP_ALT_NOT_MET = [a,b,c,d,e,f] false o_msgCARP_TAS_NOT_MET = ⊘

Tableau IV.12 SST#1b_T1

[SST#1b_T1]	Vérifie que le message CARP TAS NOT MET n'est pas généré lorsque la différence entre la vitesse TAS et la vitesse de mission IAS est plus grande que 20 noeuds (knots) et que le temps avant d'atteindre le point CRP est plus grand que 60 secondes.
Entrées :	i_rAGL = 0.0 i_rHAT = 0.0 i_TO_GREEN_LIGHT = [a] 75, [b] 75 i_rTAS = [a] 350, [b] 300 i_rIAS = [a] 300, [b] 350
Sorties anticipées :	o_msgCARP_ALT_NOT_MET = ⊘ o_msgCARP_TAS_NOT_MET = [a,b] false

Tableau IV.13 SST#1b_T2

[SST#1b_T2]	Vérifie que le message CARP TAS NOT MET n'est pas généré lorsque la différence entre la vitesse TAS et la vitesse de mission IAS est plus petite ou égale à 20 noeuds (knots) et que le temps avant d'atteindre le point CRP est plus petit ou égal à 60 secondes
Entrées :	i_rAGL = 0.0 i_rHAT = 0.0 i_TO_GREEN_LIGHT = [a,b] 45, [c,d] 60, [e,f] 45, [g,h] 60 i_rTAS = [a,c,e,g] 300, [b,d,f,h] 305 i_rIAS = [a,c,e,g] 305, [b,d,f,h] 300
Sorties anticipées :	o_msgCARP_ALT_NOT_MET = ⊘ o_msgCARP_TAS_NOT_MET = [a,b,c,d,e,f,g,h] false

Tableau IV.14 SST#1b_T3

[SST#1b_T3]	Vérifie que le message CARP TAS NOT MET n'est pas généré lorsque la différence entre la vitesse TAS et la vitesse de mission IAS est plus petite ou égale à 20 noeuds (knots) et que le temps avant d'atteindre le point CRP est plus grand que 60 secondes.
Entrées :	i_rAGL = 0.0 i_rHAT = 0.0 i_TO_GREEN_LIGHT = [a,b,c,d] 75 i_rTAS = [a,c] 300, [b,d] 305 i_rIAS = [a,c] 305, [b,d] 300
Sorties anticipées :	o_msgCARP_ALT_NOT_MET = ⊘ o_msgCARP_TAS_NOT_MET = [a,b,c,d] false

Tableau IV.15 SST#1b_T4

[SST#1b_T4]	Vérifie que le message CARP TAS NOT MET est généré lorsque la différence entre la vitesse TAS et la vitesse de mission IAS est plus grande que 20 noeuds (knots) et que le temps avant d'atteindre le point CRP est plus petit ou égal à 60 secondes.
Entrées :	i_rAGL = 0.0 i_rHAT = 0.0 i_TO_GREEN_LIGHT = [a,b] 45, [c,d] 60 i_rTAS = [a,c] 350, [b,d] 300 i_rIAS = [a,c] 300, [b,d] 350
Sorties anticipées :	o_msgCARP_ALT_NOT_MET = ⊘ o_msgCARP_TAS_NOT_MET = [a,b,c,d] true

[Tests_SST#2]

Les tests doivent être effectués sur le module CARP_Msg_Manager

Tableau IV.16 SST#2_T1

[SST#2_T1]	Vérifie que si le message NO CARP SOLUTION est présent, aucun des messages suivants ne peut l'être 20 MIN TO CARP, 10 MIN TO CARP, 1 MIN TO CARP, 5 SEC TO CARP, GREEN LIGHT, RED LIGHT .
Entrées :	i_bBaroAltAvailable = [a,b,c,d,e,f] false i_bTempAvailable = true, i_bQNHAvailable = true i_rAGL = [a,b,c,d,e,f] 1000 i_rVertDist = [a,b,c,d,e,f] 1500 i_rHAT = 1000 i_rActualTAS = 0.0 i_rMissionIAS = 0.0 i_rETE_TO_GREEN_LIGHT = [a] 1200, [b] 600, [c] 60, [d] 5, [e] 0, [e] C1:1, 0 i_Rete_before_IP_TP = 0.0 i_bIsAircraftAirbone = [a,b,c,d,e,f] true i_rETE = [a] 1200, [b] 600, [c] 60, [d] 5, [e] 0, [f] C1:1 C2:0 C3:1 C4:0 i_rTAS = 0.0 i_rIAS = 0.0 i_rTE_lat = 0.0 i_rTE_lon = 0.0 i_rPI_lat = 0.0 i_rPI_lon = 0.0 i_rIP_lat = 0.0 i_rIP_lon = 0.0 i_bMust_CARP_Terminate = false
Sorties anticipées :	o_msgCARP_ALT_NOT_MET = ⊘ o_msgCARP_TAS_NOT_MET = ⊘ o_msg20minToCARP = [a,b,c,d,e,f] false o_msg10minToCARP = [a,b,c,d,e,f] false o_msg1minToCARP = [a,b,c,d,e,f] false o_msg5secToCARP = [a,b,c,d,e,f] false o_msgGreenLight = [a,b,c,d,e,f] false o_msgRed Light = [a,b,c,d,e,f] false o_msgTE_To_IP_OUT_OF_RANGE = ⊘ o_msgTE_To_PI_OUT_OF_RANGE = ⊘ o_msgIP_To_PI_OUT_OF_RANGE = ⊘ o_msgVERT_DIST_EXCEEDS_HAT = ⊘ o_msgNOCARPSOLUTION = ⊘ o_msgCARP_ACTIVE = ⊘ o_msgAT_CARP = ⊘

[Tests_SST#3]

Les tests doivent être effectués sur le module CARP_Msg_Manager

Tableau IV.17 SST#3_T1

[SST#3_T1]	Vérifie que si l'application CARP doit terminer, les messages CARP ALT NOT MET, CARP TAS NOT MET et NO CARP SOLUTION ne doivent plus être générés.
Entrées :	i_bBaroAltAvailable = true, i_bTempAvailable = true, i_bQNHAvailable = true i_rAGL = [a,b,c] 1000 i_rVertDist = [a,b,c] 1500, [b,c] 1000 i_rHAT = [a,c] 1000, [b] 900 i_rActualTAS = 0.0 i_rMissionIAS = 0.0 i_rETE_TO_GREEN_LIGHT = [a,b,c] 55 i_Rete_before_IP_TP = 0.0 i_bIsAircraftAirbone = true i_rETE = 55.0 i_rTAS = [a,b] 0.0, [c] 350 i_rIAS = [a,b] 0.0, [c] 300 i_rTE_lat = 0.0 i_rTE_lon = 0.0 i_rPI_lat = 0.0 i_rPI_lon = 0.0 i_rIP_lat = 0.0 i_rIP_lon = 0.0 i_bMust_CARP_Terminate = [a,b,c] true
Sorties anticipées :	o_msgCARP_ALT_NOT_MET = [a,b,c] false o_msgCARP_TAS_NOT_MET = [a,b,c] false o_msg20minToCARP = ⊘ o_msg10minToCARP = ⊘ o_msg1minToCARP = ⊘ o_msg5secToCARP = ⊘ o_msgGreenLight = ⊘ o_msgRed Light = ⊘ o_msgTE_To_IP_OUT_OF_RANGE = ⊘ o_msgTE_To_PI_OUT_OF_RANGE = ⊘ o_msgIP_To_PI_OUT_OF_RANGE = ⊘ o_msgVERT_DIST_EXCEEDS_HAT = ⊘ o_msgNOCARPSOLUTION = [a,b,c] false o_msgCARP_ACTIVE = ⊘ o_msgAT_CARP = ⊘

Tableau IV.18 SST#3_T2

[SST#3_T2]	Vérifie que les messages CARP ALT NOT MET, CARP TAS NOT MET et NO CARP SOLUTION ne sont plus générés lorsque la condition les ayant activée ne s'applique plus.
Entrées :	i_bBaroAltAvailable = true, i_bTempAvailable = true, i_bQNHAvailable = true i_rAGL = [a] C1: 1000, C2: 1500, [b] 1000, [c] C1: 1000, C2: 1500, i_rVertDist = [a,b] 900, [c] 1500 i_rHAT = [a] C1: 1500, C2: 1500 [b,c] 1500 i_rActualTAS = 0.0 i_rMissionIAS = 0.0 i_rETE_TO_GREEN_LIGHT = [a] C1: 55, C2: 55 [b,c] 55 i_Rete_before_IP_TP = 0.0 i_bIsAircraftAirbone = true i_rETE = 55 i_rTAS = [a,c] 0.0, [b] C1: 350, C2: 300 i_rIAS = [a,c] 0.0, [b] C1: 300, C2: 305 i_rTE_lat = 0.0 i_rTE_lon = 0.0 i_rPI_lat = 0.0 i_rPI_lon = 0.0 i_rIP_lat = 0.0 i_rIP_lon = 0.0 i_bMust_CARP_Terminate = false
Sorties anticipées :	o_msgCARP_ALT_NOT_MET = [a] C1: true, C2: false, [b,c] ⊘ o_msgCARP_TAS_NOT_MET = [a,c] ⊘, [b] C1: true, C2: false o_msg20minToCARP = ⊘ o_msg10minToCARP = ⊘ o_msg1minToCARP = ⊘ o_msg5secToCARP = ⊘ o_msgGreenLight = ⊘ o_msgRed Light = ⊘ o_msgTE_To_IP_OUT_OF_RANGE = ⊘ o_msgTE_To_PI_OUT_OF_RANGE = ⊘ o_msgIP_To_PI_OUT_OF_RANGE = ⊘ o_msgVERT_DIST_EXCEEDS_HAT = ⊘ o_msgNOCARPSOLUTION = [a,b] ⊘ [c] C1: true, C2: false o_msgCARP_ACTIVE = ⊘ o_msgAT_CARP = ⊘

[Tests_SST#4]

Les tests doivent être effectués sur le module CARP_Status_Advisory_Msg

Tableau IV.19 SST#4a_T1

[SST#4a_T1]	Vérifie que le message 20 MIN TO CARP est seulement généré lorsqu'il reste 20 minutes et moins (mais plus de 10 minutes) avant d'atteindre le point CRP.
Entrées :	i_rETE = [a] 1201, [b] 1200, [c] 601, [d] 600
Sorties anticipées :	o_msg20minToCARP = [a] false, [b] true, [c] true, [d] false o_msg10minToCARP = ⊘ o_msg1minToCARP = ⊘ o_msg5secToCARP = ⊘ o_msgGreenLight = ⊘ o_msgRed Light = ⊘

Tableau IV.20 SST#4b_T1

[SST#4b_T1]	Vérifie que le message 10 MIN TO CARP est seulement généré lorsqu'il reste 10 minutes et moins (mais plus de 1 minute) avant d'atteindre le point CRP.
Entrées :	i_rETE = [a] 601, [b] 600, [c] 61, [d] 60
Sorties anticipées :	o_msg20minToCARP = ⊘ o_msg10minToCARP = [a] false, [b] true, [c] true, [d] false o_msg1minToCARP = ⊘ o_msg5secToCARP = ⊘ o_msgGreenLight = ⊘ o_msgRed Light = ⊘

Tableau IV.21 SST#4c_T1

[SST#4c_T1]	Vérifie que le message 1 MIN TO CARP est seulement généré lorsqu'il reste 1 minute et moins (mais plus de 5 secondes) avant d'atteindre le point CRP.
Entrées :	i_rETE = [a] 61, [b] 60, [c] 6, [d] 5
Sorties anticipées :	o_msg20minToCARP = ⊘ o_msg10minToCARP = ⊘ o_msg1minToCARP = [a] false, [b] true, [c] true, [d] false o_msg5secToCARP = ⊘ o_msgGreenLight = ⊘ o_msgRed Light = ⊘

Tableau IV.22 SST#4d_T1

[SST#4d_T1]	Vérifie que le message 5 SEC TO CARP est seulement généré lorsqu'il reste 5 secondes et moins (mais plus de 0 seconde) avant d'atteindre le point CRP.
Entrées :	i_rETE = [a] 6, [b] 5, [c] 1, [d] 0
Sorties anticipées :	o_msg20minToCARP = ⊘ o_msg10minToCARP = ⊘ o_msg1minToCARP = ⊘ o_msg5secToCARP = [a] false, [b] true, [c] true, [d] false o_msgGreenLight = ⊘ o_msgRed Light = ⊘

Tableau IV.23 SST#4e_T1

[SST#4e_T1]	Vérifie que le message GREEN LIGHT est seulement généré le point CRP est atteint et désactivé quand le point XTE (RED LIGHT) est atteint
Entrées :	i_rETE = [a] 1, [b] 0, [c] C1: 900, C2: 300, C3: 50, C4: 1 [d] 0
Sorties anticipées :	o_msg20minToCARP = ⊘ o_msg10minToCARP = ⊘ o_msg1minToCARP = ⊘ o_msg5secToCARP = ⊘ o_msgGreenLight = [a] false, [b] true, [c] true, [d] false o_msgRed Light = ⊘

Tableau IV.24 SST#4f_T1

[SST#4f_T1]	Vérifie que le message RED LIGHT est seulement généré lorsque le point XTE est atteint.
Entrées :	i_rETE = [a] 1, [b] 0, [c] 1, [d] 0
Sorties anticipées :	o_msg20minToCARP = ⊘ o_msg10minToCARP = ⊘ o_msg1minToCARP = ⊘ o_msg5secToCARP = ⊘ o_msgGreenLight = ⊘ o_msgRed Light = [a] false, [b] false, [c] false, [d] true

[Tests_SST#5]

Les tests doivent être effectués sur le module CARP_DataEntry_Advisory_Msg

Tableau IV.25 SST#5a_T1

[SST#5a_T1]	Vérifie que le message TE TO IP OUT OF RANGE est généré seulement lorsque la distance entre les points TE et IP est plus grande que 50 nm.
Entrées :	i_rDist_TE_IP = [a] 51, [b] 50 i_rDist_TE_PI = 0.0 i_rDist_IP_PI = 0.0 i_rVertDist = 0.0 i_rHat = 0.0
Sorties anticipées :	o_msgTE_To_IP_OUT_OF_RANGE = [a] true, [b] false o_msgTE_To_PI_OUT_OF_RANGE = ⊘ o_msgIP_To_PI_OUT_OF_RANGE = ⊘ o_msgVERT_DIST_EXCEEDS_HAT = ⊘

Tableau IV.26 SST#5b_T1

[SST#5b_T1]	Vérifie que le message TE TO PI OUT OF RANGE est généré seulement lorsque la distance entre les points TE et PI est plus grande que 20 nm.
Entrées :	i_rDist_TE_IP = 0.0 i_rDist_TE_PI = [a] 21, [b] 20 i_rDist_IP_PI = 0.0 i_rVertDist = 0.0 i_rHat = 0.0
Sorties anticipées :	o_msgTE_To_IP_OUT_OF_RANGE = ⊘ o_msgTE_To_PI_OUT_OF_RANGE = [a] true, [b] false o_msgIP_To_PI_OUT_OF_RANGE = ⊘ o_msgVERT_DIST_EXCEEDS_HAT = ⊘

Tableau IV.27 SST#5c_T1

[SST#5c_T1]	Vérifie que le message IP TO PI OUT OF RANGE est généré seulement lorsque la distance entre les points IP et PI est plus grande que 30 nm.
Entrées :	i_rDist_TE_IP = 0.0 i_rDist_TE_PI = 0.0 i_rDist_IP_PI = [a] 31, [b] 30 i_rVertDist = 0.0 i_rHat = 0.0
Sorties anticipées :	o_msgTE_To_IP_OUT_OF_RANGE = ⊘ o_msgTE_To_PI_OUT_OF_RANGE = ⊘ o_msgIP_To_PI_OUT_OF_RANGE = [a] true, [b] false o_msgVERT_DIST_EXCEEDS_HAT = ⊘

Tableau IV.28 SST#5d_T1

[SST#5d_T1]	Vérifie que le message VERT DIST EXCEEDS HAT est généré seulement lorsque la distance verticale VERT DIST dépasse la hauteur Mission Height above point of impact MISSION HAT.
Entrées :	i_rDist_TE_IP = 0.0 i_rDist_TE_PI = 0.0 i_rDist_IP_PI = 0.0 i_rVertDist = [a] 1000, [b] 900, [c] 900 i_rHat = [a] 900, [b] 1000, [c] 900
Sorties anticipées :	o_msgTE_To_IP_OUT_OF_RANGE = ⊘ o_msgTE_To_PI_OUT_OF_RANGE = ⊘ o_msgIP_To_PI_OUT_OF_RANGE = ⊘ o_msgVERT_DIST_EXCEEDS_HAT = [a] true, [b,c] false

[Tests_SST#6]

Les tests doivent être effectués sur le module CARP_NoCarpSolution_Msg

Tableau IV.29 SST#6a_T1

[SST#6a_T1]	Vérifie que le message NO CARP SOLUTION est généré seulement lorsque l'entrée CARP ACTIVE est présente, mais que la Baro Altitude n'est pas disponible.
Entrées : **Entrées :**	i_bCarpActive = [a,b] false, [c,d] true i_bBaroAltAvailable = [a] true, [b] false, [c] true, [d] false i_bTempAvailable = true i_bQNHAvailable = true i_rAGL = 0.0 i_rVertDist = 0.0 i_rHat = 0.0 i_rActualTAS = 0.0 i_rMissionIAS = 0.0 i_rETE_TO_GREEN_LIGHT = 45.0 i_Rete_before_IP_TP = 1.0
Sorties anticipées :	o_msgNOCARPSOLUTION = [a] false, [b] false, [c] false, [d] true

Tableau IV.30 SST#6b_T1

[SST#6b_T1]	Vérifie que le message NO CARP SOLUTION est généré seulement lorsque l'entrée CARP ACTIVE est présente, mais que température n'est pas disponible.
Entrées :	i_bCarpActive = [a,b] false, [c,d] true i_bBaroAltAvailable = true i_bTempAvailable = [a] true, [b] false, [c] true, [d] false i_bQNHAvailable = true i_rAGL = 0.0 i_rVertDist = 0.0 i_rHat = 0.0 i_rActualTAS = 0.0 i_rMissionIAS = 0.0 i_rETE_TO_GREEN_LIGHT = 45.0 i_Rete_before_IP_TP = 1.0
Sorties anticipées :	o_msgNOCARPSOLUTION = [a] false, [b] false, [c] false, [d] true

Tableau IV.31 SST#6c_T1

[SST#6c_T1]	Vérifie que le message NO CARP SOLUTION est généré seulement lorsque l'entrée CARP ACTIVE est présente, mais que la Baro Altitude Correction n'est pas disponible.
Entrées :	i_bCarpActive = [a,b] false, [c,d] true i_bBaroAltAvailable = true i_bTempAvailable = true i_bQNHAvailable = [a] true, [b] false, [c] true, [d] false i_rAGL = 0.0 i_rVertDist = 0.0 i_rHat = 0.0 i_rActualTAS = 0.0 i_rMissionIAS = 0.0 i_rETE_TO_GREEN_LIGHT = 45.0 i_Rete_before_IP_TP = 1.0
Sorties anticipées :	o_msgNOCARPSOLUTION = [a] false, [b] false, [c] false, [d] true

Tableau IV.32 SST#6d_T1

[SST#6d_T1]	Vérifie que le message NO CARP SOLUTION est généré seulement lorsqu'il reste moins d'une minute avant d'atteindre le point CRP et que l'altitude actuelle (AGL) est plus petite que la distance verticale (Vert Dist).
Entrées :	i_bCarpActive = false i_bBaroAltAvailable = ⊘ i_bTempAvailable = ⊘ i_bQNHAvailable = ⊘ i_rAGL = [a,b,d,e] 1000, [c,f] 900 i_rVertDist = [a,d] 900, [b,c,e,f] 1000 i_rHat = 1000 i_rActualTAS = 0.0 i_rMissionIAS = 0.0 i_rETE_TO_GREEN_LIGHT = [a,b,c] 65, [d,e,f] 45 i_Rete_before_IP_TP = 1.0
Sorties anticipées :	o_msgNOCARPSOLUTION = [a,b,c,d,e] false, [f] true

Tableau IV.33 SST#6e_T1

[SST#6e_T1]	Vérifie que le message NO CARP SOLUTION est généré seulement lorsqu'il reste moins d'une minute avant d'atteindre le point CRP et que la différence d'altitude entre l'altitude actuelle (AGL) et la hauteur Mission Height above point of impact (MISSION HAT) est plus grande ou égale à 1000 pieds.
Entrées :	i_bCarpActive = false i_bBaroAltAvailable = ⊘ i_bTempAvailable = ⊘ i_bQNHAvailable = ⊘ i_rAGL = [a,d] 1900, [b,e] 2000, [c,f] 2100 i_rVertDist = 1000 i_rHat = [a,b,c,d,e,f] 1000 i_rActualTAS = 0.0 i_rMissionIAS = 0.0 i_rETE_TO_GREEN_LIGHT = [a,b,c] 65, [d,e,f] 45 i_Rete_before_IP_TP = 1.0
Sorties anticipées :	o_msgNOCARPSOLUTION = [a,b,c,d] false, [e,f] true

Tableau IV.34 SST#6f_T1

[SST#6f_T1]	Vérifie que le message NO CARP SOLUTION est généré seulement lorsqu'il reste moins d'une minute avant d'atteindre le point CRP et que la différence entre la vitesse TAS et la vitesse IAS est plus grande ou égale à 50 knots.
Entrées :	i_bCarpActive = false i_bBaroAltAvailable = ⊘ i_bTempAvailable = ⊘ i_bQNHAvailable = ⊘ i_rAGL = 0.0 i_rVertDist = 0.0 i_rHat = 0.0 i_rActualTAS = [a,d] 90, [b,e] 100, [c,f] 110, [g,h] 50 i_rMissionIAS = [a,b,c,d,e,f] 50, [g] 90, [h] 100 i_rETE_TO_GREEN_LIGHT = [a,b,c] 65, [d,e,f,g,h] 45 i_Rete_before_IP_TP = 1.0
Sorties anticipées :	o_msgNOCARPSOLUTION = [a,b,c,d,g] false, [e,f,h] true

Tableau IV.35 SST#6g_T1

[SST#6g_T1]	Vérifie que lorsque le message NO CARP SOLUTION est généré parce l'entrée CARP ACTIVE est présente, mais que la Baro Altitude, la Baro Altitude Correction ou la température n'est pas disponible, celui-ci peut être supprimé s'il reste 1 minute et plus avant d'atteindre le point CRP et la Baro Altitude, la Baro Altitude Correction ou la température redevient disponible.
Entrées :	i_bCarpActive = [a,b,c,d,e,f] true i_bBaroAltAvailable = [a,d] C1:false, C2:true, [b,c,e,f] true i_bTempAvailable = [a,c,d,f] true, [b,e] C1:false, C2:true i_bQNHAvailable = [a,b,d,e] true, [c,f] C1:false, C2:true i_rAGL = 0.0 i_rVertDist = 0.0 i_rHat = 0.0 i_rActualTAS = 0.0 i_rMissionIAS = 0.0 i_rETE_TO_GREEN_LIGHT = 75.0 i_Rete_before_IP_TP = [a,b,c] 60, [d,e,f] 61
Sorties: anticipées:	o_msgNOCARPSOLUTION = [a,b,c] C1:true, C2:true [d,e,f] C1:true, C2:false

Tableau IV.36 SST#6h_T1

[SST#6h_T1]	Vérifie que lorsque le message NO CARP SOLUTION est généré parce qu'il reste moins d'une minute avant d'atteindre le point CRP et que l'altitude actuelle (AGL) est plus petite que la distance verticale (Vert Dist), celui-ci peut être supprimé s'il reste plus de 30 secondes avant d'atteindre le point CRP et que l'altitude actuelle (AGL) devient plus grande ou égale que la distance verticale (Vert Dist).
Entrées :	i_bCarpActive = false i_bBaroAltAvailable = ⊘ i_bTempAvailable = ⊘ i_bQNHAvailable = ⊘ i_rAGL = [a,d] C1:900, C2:999, [b,e] C1:900, C2:1000 [c,f] C1:900, C2:1001 i_rVertDist = [a,b,c,d,e,f] 1000 i_rHat = 1000 i_rActualTAS = 0.0 i_rMissionIAS = 0.0 i_rETE_TO_GREEN_LIGHT = [a,b,c] 30, [d,e,f] 31 i_Rete_before_IP_TP = 1.0
Sorties: anticipées:	o_msgNOCARPSOLUTION = [a,b,c,d] C1:true, C2:true [e,f] C1:true, C2:false

Tableau IV.37 SST#6h_T2

[SST#6h_T2]	Vérifie que lorsque le message NO CARP SOLUTION est généré parce qu'il reste moins d'une minute avant d'atteindre le point CRP et que la différence d'altitude entre l'altitude actuelle (AGL) et la hauteur Mission Height above point of impact (MISSION HAT) est plus grande ou égale à 1000 pieds, celui-ci peut être supprimé si il reste plus de 30 secondes avant d'atteindre le point CRP et que la différence d'altitude entre l'altitude actuelle (AGL) et la hauteur Mission Height above point of impact (MISSION HAT) devienne plus petite que 1000 pieds.
Entrées :	i_bCarpActive = false i_bBaroAltAvailable = ⊘ i_bTempAvailable = ⊘ i_bQNHAvailable = ⊘ i_rAGL = [a,d] C1:2100, C2:2001, [b,e] C1:2100, C2:2000 [c,f] C1:2100, C2:1999 i_rVertDist = 1000 i_rHat = [a,b,c,d,e,f] 1000 i_rActualTAS = 0.0 i_rMissionIAS = 0.0 i_rETE_TO_GREEN_LIGHT = [a,b,c] 30, [d,e,f] 31 i_Rete_before_IP_TP = 1.0
Sorties: anticipées:	o_msgNOCARPSOLUTION = [a,b,c,d,e] C1:true, C2:true [f] C1:true, C2:false

Tableau IV.38 SST#6h_T3

[SST#6h_T3]	Vérifie que lorsque le message NO CARP SOLUTION est généré parce qu'il reste moins d'une minute avant d'atteindre le point CRP et que la différence entre la vitesse TAS et la vitesse IAS est plus grande ou égale à 50 knots, celui-ci peut être supprimé si il reste plus de 30 secondes avant d'atteindre le point CRP et que la différence entre la vitesse TAS et la vitesse IAS devienne plus petite que 50 knots.
Entrées :	i_bCarpActive = false i_bBaroAltAvailable = ⊘ i_bTempAvailable = ⊘ i_bQNHAvailable = ⊘ i_rAGL = 0.0 i_rVertDist = 0.0 i_rHat = 0.0 i_rActualTAS = [a,d] C1:110, C2:105, [b,e] C1:110, C2:100 [c,f] C1:110, C2:99 i_rMissionIAS = [a,b,c,d,e,f] 50 i_rETE_TO_GREEN_LIGHT = [a,b,c] 30, [d,e,f] 31 i_Rete_before_IP_TP = 1.0
Sorties: anticipées:	o_msgNOCARPSOLUTION = [a,b,c,d,e] C1:true, C2:true [f] C1:true, C2:false

Tableau IV.39 SST#6h_T4

[SST#6h_T4]	Vérifie que lorsque le message NO CARP SOLUTION est généré parce qu'il reste moins d'une minute avant d'atteindre le point CRP et que la différence entre la vitesse TAS et la vitesse IAS est plus grande ou égale à 50 knots, et que la différence d'altitude entre l'altitude actuelle (AGL) et la hauteur Mission Height above point of impact (MISSION HAT) est plus grande ou égale à 1000 pieds et que l'entrée CARP ACTIVE est présente, mais que la Baro Altitude, la Baro Altitude Correction ou la température n'est pas disponible, celui-ci (NO CARP SOLUTION msg) peut être supprimé si il reste 30 secondes et plus avant d'atteindre le point CRP et que la différence entre la vitesse TAS et la vitesse IAS devienne plus petite que 50 knots et que la différence d'altitude entre l'altitude actuelle (AGL) et la hauteur Mission Height above point of impact (MISSION HAT) devienne plus petite que 1000 pieds et si il reste 1 minute et plus avant d'atteindre le point IP ou TP et la Baro Altitude, la Baro Altitude Correction ou la température redevient disponible.
Entrées :	i_bCarpActive = true i_bBaroAltAvailable= [a,h]C1:false, C2:false, [b,i]C1:false, C2:true [c,j] C1:true, C2:false, C3:true [d,k] C1:false, C2:false, C3:true [e,l,g,n] C1:false, C2:false, C3:false [f,m] C1:false, C2:true, C3:true i_bTempAvailable = true i_bQNHAvailable = true i_rAGL = [a,h] C1:2100, C2:2001, [b,i] C1:2100, C2:1999 [c,g,j,n] C1:1999, C2:2100, C3:1999 [d,f,k,m] C1:2100, C2:2100, C3:1999 [e,l] C1:2100, C2:2100, C3:2100 i_rVertDist = 1500 i_rHat = 1000 i_rActualTAS = [a,h] C1:110, C2:100, [b,i] C1:110, C2:99 [c,j] C1:110, C2:110, C3:99 [d,e,k] C1:99, C2:110, C3:99 [f,g] C1:99, C2:110, C3:99, [l,m,n] 50 i_rMissionIAS = [a,b,c,d,e,f,g,h,i,k,l] 50 [m] C1:99, C2:110, C3:99 [n] C1:99, C2:99, C3:99 i_rETE_TO_GREEN_LIGHT = [a,b,c,d,e,f,g] 30, [h,i,j,k,l,m,n] 31 i_Rete_before_IP_TP = [a,b,c,d,e,f,g] 60, [h,i,j,k,l,m,n] 61
Sorties: anticipées:	o_msgNOCARPSOLUTION = [a,b,h] C1:true, C2:true [c,d,e,f,g,l,n] C1:true, C2:true, C3:true [j,k,m] C1:true, C2:true, C3:false [i] C1:true, C2:false

[Tests_SST#7]

Les tests doivent être effectués sur le module CARP_Active_Annunciator_Msg

Tableau IV.40 SST#7_T1

[SST#7_T1]	Vérifie que la sortie discrète CARP ACTIVE est vraie seulement si l'avion est en vol et qu'il reste 20 minutes et moins avant d'atteindre le point CRP.
Entrées :	i_bIsAircraftAirbone = [a,b,c] false, [d,e,f] true i_rETE_TO_GREEN_LIGHT = [a,d] 1201, [b,e] 1200, [c,f] 600
Sorties anticipées :	o_msgCARP_ACTIVE = [a,b,c,d] false, [e,f] true

[Tests_SST#8]

Les tests doivent être effectués sur le module CARP_AtCarp_Annunciator_Msg

Tableau IV.41 SST#8_T1

[SST#8_T1]	Vérifie que la sortie discrète CARP ACTIVE est mise à vraie 5 seconde avant d'atteindre le point CRP, et mise à fausse lorsque le point XTE (Red Light) est atteint.
Entrées :	i_bIsRed_Light = [a,b,c] false, [d,e] true i_rETE_TO_GREEN_LIGHT = [a] 6, [b] 5, [c] 0, [d] 6, [e] 0
Sorties anticipées :	o_msgCARP_ACTIVE = [a,d,e] false, [b,c] true

[Tests_SST#9]

Les tests doivent être effectués sur le module CARP_Calculate_Distance

Tableau IV.42 SST#9_T1

[SST#9_T1]	Vérifie que la bonne distance est calculée, peu importe, les coordonnées entrées.
Entrées :	i_Rpoint_1_lat = [a] 35.0, [b] 35.0, [c] 45.0, [d] 55.0 [e] -15.0, [f] -40.0, [g] -25.0, [h] -44.0, [i] 45.5 i_Rpoint_1_lon = [a] 65.0, [b] 65.0, [c] -85.0, [d] -135.0 [e] 15.0, [f] 62.0, [g] -18.0, [h] -64.0, [i] 73.583 i_Rpoint_2_lat = [a] 35.0, [b] 45.0, [c] -25.0, [d] -5.0 [e] -45.0, [f] 75.0, [g] -15.0, [h] 83.0, [i] 43.65 i_Rpoint_2_lon = [a] 65.0, [b] 90.0, [c] 75.0, [d] -165.0 [e] -180.0, [f] 0.0, [g] -15.0, [h] 48.0, [i] 79.383
Sorties: anticipées:	o_Rdistance = [a] 0.0, [b] 1287.5, [c] 9247.8, [d] 3892.7 [e] 7100.9, [f] 7303.6, [g] 622.63, [h] 8166.1 [i] 271.24